Theory of Computation

Raju Hazari

Department of Computer Science and Engineering
National Institute of Technology Calicut

August 9, 2021

Books and References

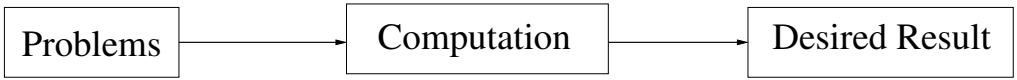
1. Michael Sipser, *Introduction to the Theory of Computation*, third edition, 2005.
2. John E Hopcroft, Rajeev Motwani, Jeffrey D Ullman, *Introduction to Automata Theory, Languages, and Computation*, third edition, Prentice Hall, 2007.
3. Dexter C Kozen, *Automata and Computability*, Springer, 1997.
4. John C Martin, *Introduction to Languages and the Theory of Computation*, third edition, Tata McGraw-Hill, 2003.
5. Peter Linz , *An Introduction to Formal Languages and Automata*, sixth edition, 2016.

Introduction

- One of the most fundamental course of computer science.
- It is mainly about what kind of things can you really compute mechanically, how fast and how much space does it take to do so.

What is Computation?

- Today, what we understand as computation is that you write some programs and you run the programs in computers and in that way you carry out a computation.
- From the word "Computation", we understand that there are some problems which need to compute to get the desired result.



What is Computation?

- Can we compute anything ?

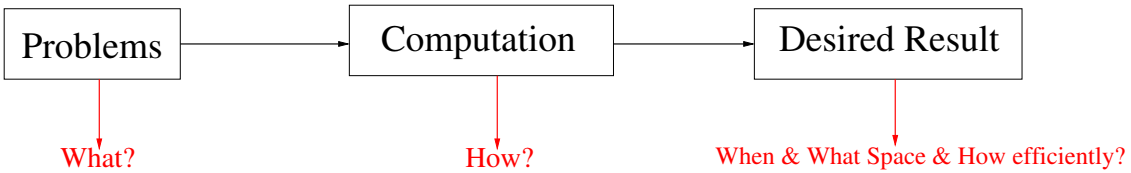
What is Computation?

- Can we compute anything ?
  - No.
  - So, there are few questions.

What is Computation?

- Can we compute anything ?
  - No.
  - So, there are few questions.

1. What kind of problem can be computed ?
2. How a problem can be computed ?
   - Means, what is the method by which a problem can be computed to achieve the desired result.
3. When & What Space & How Efficiently can a problem can be computed ?



Different areas in Theory of Computation

- All the 3 questions will be answer in Theory of Computation, which means the field Theory of Computation is divided into three parts.
  1. "What" kind of problem can be computed, which means COMPUTABILITY THEORY.
  2. "How" a problem can be computed, which is AUTOMATA THEORY.
  3. When & What Space & How Efficiently can a problem can be computed, means it is COMPLEXITY THEORY.
- All these three are linked together by one question, which is– *"what are the fundamental capabilities and limitation of a computational model ?"*
- In today's world, "Computational model" means it is computer.

Different areas in Theory of Computation

- COMPUTABILITY THEORY: Here, we are going to deal with what kind of problem are computable and which are not computable.
- AUTOMATA THEORY: Here, we will deal with the mathematical models or the abstract machine model, which can performed the computation.
- COMPLEXITY THEORY: Here, we will deal with the efficiency of those mathematical model. In how much time, and what space can they compute a particular problem.

COMPUTABILITY THEORY

- The major question:
  - What is computable problem ?
  - What is non-computable problem ?

COMPUTABILITY THEORY

- The major question:
  - What is computable problem ?
  - What is non-computable problem ?
- **Computable Problem**: A problem is computable means there exists an algorithm that computes an answer to any instance of the problem in a finite number of simple steps.
  - Example: Computing the greatest common divisor of a pair of integers.
- **Non-computable Problem**: A non-computable is a problem for which there is no algorithm that can be used to solve it.
  - Example: Halting Problem ("Given a Turing machine in an arbitrary configuration will it eventually halt ?")

AUTOMATA THEORY

- If some thing is computable, then 'how' can be computabled, is what we will discuss in "**Automata Theory**".
- How a problem can be computed using abstract machine models or abstract mathematical models, and which are those mathematical models.
- As it is described in 1930s, by Alan Turing and Alonzo church, it was **Turing Machine**.
- Turing machine is the most powerful mathematical model, which can performed the computation on anything which is computable.

AUTOMATA THEORY

- Turing machine is one of the abstract machine model. But, there are some other models, which were recognize by different formal language.
- For every particular formal language there is one computational model or one abstract model.
- There is a one specific formal language for turing machine also, which is known as **Recursively Enumerable language** or **Unrestricted grammar** or **Phrase structure grammar**.

AUTOMATA THEORY

- Another abstract machine model is **Linear Bounded Automata**. It is restricted form of turing machine with only few restriction.
- Next one, even it is more restricted form, which is **Pushdown Automata**.
- The simplest and most restricted form of all is **Finite State Machine** or **Finite Automata**.
- All these four abstract machine models have their own formal language.
  - Turing Machine: Recursively enumerable language
  - Linear Bounded Automata: Context-sensitive language
  - Pushdown Automata: Context-free language
  - Finite Automata: Regular language

COMPLEXITY THEORY

- Computational problems come in different varieties; some are easy, and some are hard. For example–
  - Sorting problem: Arrange a list of numbers in ascending order.
  - Scheduling problem: Find a schedule of classes for the entire university in such a way that no two classes take place in the same room at the same time.
- The scheduling problem seems to be much harder than the sorting problem.

"What makes some problems computationally hard and other easy ?"

- This is the central question of complexity theory.
- We don't know the answer to it, through it has been intensively researched for over 40 years.

COMPLEXITY THEORY

- One important achievement of complexity theory thus far, researchers have discovered an elegant scheme for **classifying problems according to their computational difficulty.**
- Using this scheme, we can demonstrate a method for giving evidence that certain problems are computationally hard, even if we are unable to prove that they are.
- Classification of problems:
  - P problem
  - NP problem
  - NP Complete problem
  - NP Hard problem
  - PSPACE problem
  - PSPACE Complete problem

Strings, Languages and Chomsky hierarchy

Raju Hazari

Department of Computer Science and Engineering
National Institute of Technology Calicut

August 18, 2021

Alphabet

- Here, we introduced the most important definitions of terms that pervade the theory of automata.
- These concepts include the "**alphabet**", "**strings**" and "**language**".

Alphabet

- Here, we introduced the most important definitions of terms that pervade the theory of automata.
- These concepts include the "**alphabet**", "**strings**" and "**language**".

- What is symbol ?
  - **Symbol:** 0,1 are the symbols. Similarly, a, b are also symbols

Alphabet

- Here, we introduced the most important definitions of terms that pervade the theory of automata.
- These concepts include the "**alphabet**", "**strings**" and "**language**".

- What is symbol ?
  - **Symbol:** 0,1 are the symbols. Similarly, a, b are also symbols

- What is Alphabet ?
  - **Alphabet:** A *alphabet* is a finite, nonempty set of symbols. Conventionally, we use the symbol Σ for an alphabet.
    - $\Sigma = \{0, 1\}$, the binary alphabet
    - $\Sigma = \{a, b, \cdots, z\}$, the set of all lower-case letter
    - The set of all ASCII characters, or the set of all printable ASCII characters.

Strings

- What is a String ?
  - **String:** A *string* (or sometimes word) is a finite sequence of symbols chosen from some alphabet.
    - ★ For example, 01101 is a string from the binary alphabet $\Sigma = \{0, 1\}$.
    - ★ 0110 is another string chosen from this alphabet.
  - **Empty String:** The *empty string* is the string with zero occurrences of symbols. This string denoted by ϵ.
  - **Length of a String:** The number of positions for symbols in the string.
    - ★ For instance, 01101 has length 5.
    - ★ The standard notation for the length of a string w is $|w|$. For example, $|011| = 3$ and $|\epsilon| = 0$

Strings

- **Powers of an Alphabet:**
  - If Σ is an alphabet, we can express the set of all strings of a certain length from that alphabet by using an exponential notation.
  - We define Σ^k to be the set of strings of length k, each of whose symbols is in Σ.
  - For Example: If $\Sigma = \{a, b, c\}$ then $\Sigma^1 = \{a, b, c\}$,
    $\Sigma^2 = \{aa, ab, ac, ba, bb, bc, ca, cb, cc\}$,
    $\Sigma^3 = \{aaa, aab, aac, \cdots, cca, ccb, ccc\}$

Strings

- What is Σ^* ?
  - The set of all strings over an alphabet Σ is conventionally denoted by Σ^*.
  - For instance, if $\Sigma = \{0, 1\}$, then $\Sigma^* = \{\epsilon, 0, 1, 00, 01, 10, 11, 000, \cdots\}$.
  - So, in this case, Σ^* is going to be the set of all finite binary strings.
  - $\Sigma^* = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \cdots$
  - So, Σ^* is an infinite set and all its members are finite length, means we do not allow in Σ^* to have an infinitely long string.
- Sometimes, we exclude the empty string from the set of strings. The set of nonempty strings from alphabet Σ is denoted by Σ^+
  - $\Sigma^+ = \Sigma^1 \cup \Sigma^2 \cup \Sigma^3 \cup \cdots$
  - $\Sigma^* = \Sigma^+ \cup \{\epsilon\}$

Language

- A formal language L over the alphabet Σ is a subset of Σ^*, means, $L \subseteq \Sigma^*$.
- For example, if $\Sigma = \{0, 1\}$, then {01, 010, 011, 10110, 110011, 110110} is a set of strings over the alphabet {0, 1} and clearly this is a subset of $\{0, 1\}^*$, means $\{01, 010, 011, 10110, 110011, 110110\} \subseteq \Sigma^*$.
- So, this is a language L. We can say, this is a example of a binary language.
- **Example:** $L_1 = \{x \in \{0, 1\}^* | \ x$ has even number of 0's and even number of 1's $\}$
  - L_1 is an example of a language over alphabet {0, 1}, which is infinite, but which is not entire $\{0, 1\}^*$
  - So, the language L_1 is a proper subset of $\{0, 1\}^*$, means $L_1 \subset \{0, 1\}^*$

Grammar

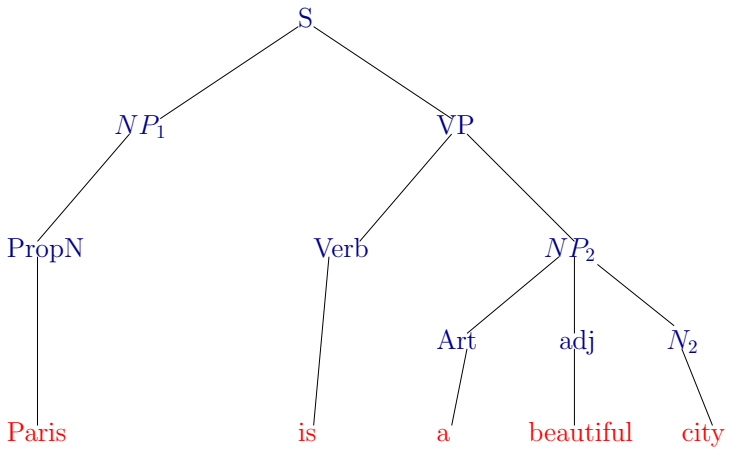
- In 1959, Noam Chomsky try to define that–
  - What is a mathematical model of a grammar ?
- The motivation was to study parsing in natural languages like English.
  - He tried to look at the parse tree of the natural language sentences and then tried to define "What is a grammar ?"

Grammar

- Let us take a sentence " **Paris is a beautiful city** " and see how they can be parsed.

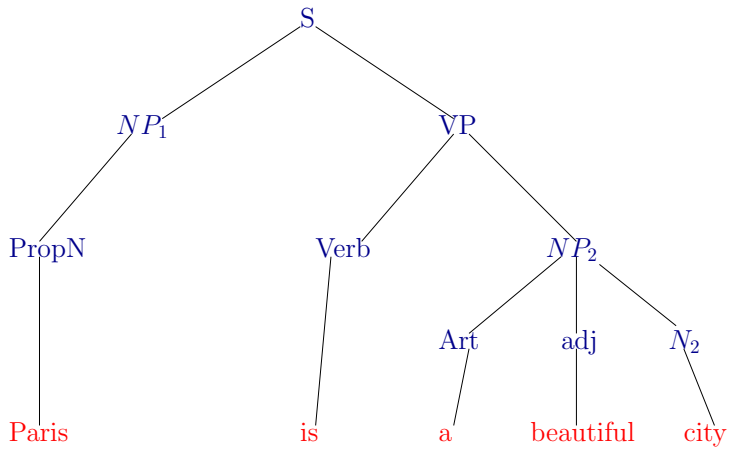
Grammar

- Let us take a sentence " **Paris is a beautiful city** " and see how they can be parsed.



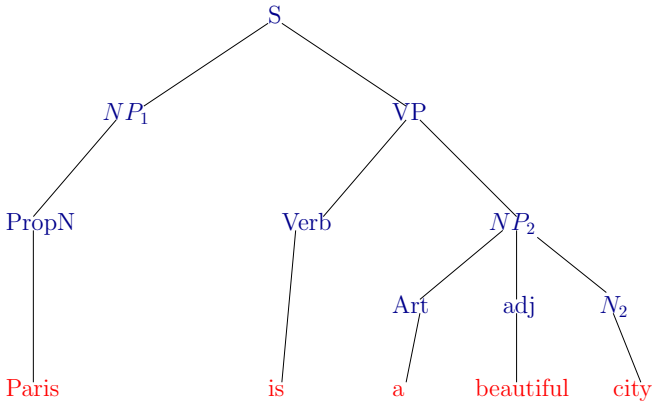
Grammar

- Let us take a sentence " **Paris is a beautiful city** " and see how they can be parsed.



- The parse tree are the syntax tree for the grammar.
- We also call it as a derivation tree.
  - If we go from sentence symbol to the sentence, it is a derivation.
  - If we go from the sentence to the sentence symbol, trying to construct the tree in the bottom up manner, it is parsing.

Grammar



- Let us look at the rules here:
  - $\langle S \rangle \rightarrow \langle NP_1 \rangle \langle VP \rangle$
  - $\langle NP_1 \rangle \rightarrow \langle PropN \rangle$
  - $\langle PropN \rangle \rightarrow$ Paris
  - $\langle VP \rangle \rightarrow \langle Verb \rangle \langle NP_2 \rangle$
  - $\langle Verb \rangle \rightarrow$ is
  - $\langle NP_2 \rangle \rightarrow \langle Art \rangle \langle adj \rangle \langle N_2 \rangle$
  - $\langle Art \rangle \rightarrow$ a
  - $\langle adj \rangle \rightarrow$ beautiful
  - $\langle N_2 \rangle \rightarrow$ city

Grammar

- These are called rules or production which are sometimes called production rules.
- Using these rules we can derive the sentence "**Paris is a beautiful city**"
- How is the derivation done ?

Grammar

- These are called rules or production which are sometimes called production rules.
- Using these rules we can derive the sentence "**Paris is a beautiful city**"
- How is the derivation done ?

- The rules are:
  - $\langle S \rangle \rightarrow \langle NP_1 \rangle \langle VP \rangle$
  - $\langle NP_1 \rangle \rightarrow \langle PropN \rangle$
  - $\langle PropN \rangle \rightarrow$ Paris
  - $\langle VP \rangle \rightarrow \langle Verb \rangle \langle NP_2 \rangle$
  - $\langle Verb \rangle \rightarrow$ is
  - $\langle NP_2 \rangle \rightarrow \langle Art \rangle \langle adj \rangle \langle N_2 \rangle$
  - $\langle Art \rangle \rightarrow$ a
  - $\langle adj \rangle \rightarrow$ beautiful
  - $\langle N_2 \rangle \rightarrow$ city
- Derivation:

Grammar

- These are called rules or production which are sometimes called production rules.
- Using these rules we can derive the sentence "**Paris is a beautiful city**"
- How is the derivation done ?

- The rules are:
  - $\langle S \rangle \rightarrow \langle NP_1 \rangle \langle VP \rangle$
  - $\langle NP_1 \rangle \rightarrow \langle PropN \rangle$
  - $\langle PropN \rangle \rightarrow$ Paris
  - $\langle VP \rangle \rightarrow \langle Verb \rangle \langle NP_2 \rangle$
  - $\langle Verb \rangle \rightarrow$ is
  - $\langle NP_2 \rangle \rightarrow \langle Art \rangle \langle adj \rangle \langle N_2 \rangle$
  - $\langle Art \rangle \rightarrow$ a
  - $\langle adj \rangle \rightarrow$ beautiful
  - $\langle N_2 \rangle \rightarrow$ city

- Derivation:
  - $\langle S \rangle \Rightarrow \langle NP_1 \rangle \langle VP \rangle$

Grammar

- These are called rules or production which are sometimes called production rules.
- Using these rules we can derive the sentence "**Paris is a beautiful city**"
- How is the derivation done ?

- The rules are:
  - $\langle S \rangle \rightarrow \langle NP_1 \rangle \langle VP \rangle$
  - $\langle NP_1 \rangle \rightarrow \langle PropN \rangle$
  - $\langle PropN \rangle \rightarrow$ Paris
  - $\langle VP \rangle \rightarrow \langle Verb \rangle \langle NP_2 \rangle$
  - $\langle Verb \rangle \rightarrow$ is
  - $\langle NP_2 \rangle \rightarrow \langle Art \rangle \langle adj \rangle \langle N_2 \rangle$
  - $\langle Art \rangle \rightarrow$ a
  - $\langle adj \rangle \rightarrow$ beautiful
  - $\langle N_2 \rangle \rightarrow$ city

- Derivation:
  - $\langle S \rangle \Rightarrow \langle NP_1 \rangle \langle VP \rangle$
  - $\Rightarrow \langle PropN \rangle \langle VP \rangle$

Grammar

- These are called rules or production which are sometimes called production rules.
- Using these rules we can derive the sentence "**Paris is a beautiful city**"
- How is the derivation done ?

- The rules are:
  - $\langle S \rangle \rightarrow \langle NP_1 \rangle \langle VP \rangle$
  - $\langle NP_1 \rangle \rightarrow \langle PropN \rangle$
  - $\langle PropN \rangle \rightarrow$ Paris
  - $\langle VP \rangle \rightarrow \langle Verb \rangle \langle NP_2 \rangle$
  - $\langle Verb \rangle \rightarrow$ is
  - $\langle NP_2 \rangle \rightarrow \langle Art \rangle \langle adj \rangle \langle N_2 \rangle$
  - $\langle Art \rangle \rightarrow$ a
  - $\langle adj \rangle \rightarrow$ beautiful
  - $\langle N_2 \rangle \rightarrow$ city

- Derivation:
  - $\langle S \rangle \Rightarrow \langle NP_1 \rangle \langle VP \rangle$
  - $\Rightarrow \langle PropN \rangle \langle VP \rangle$
  - $\Rightarrow$ **Paris** $\langle VP \rangle$

Grammar

- These are called rules or production which are sometimes called production rules.
- Using these rules we can derive the sentence "**Paris is a beautiful city**"
- How is the derivation done ?

- The rules are:
  - $\langle S \rangle \rightarrow \langle NP_1 \rangle \langle VP \rangle$
  - $\langle NP_1 \rangle \rightarrow \langle PropN \rangle$
  - $\langle PropN \rangle \rightarrow$ Paris
  - $\langle VP \rangle \rightarrow \langle Verb \rangle \langle NP_2 \rangle$
  - $\langle Verb \rangle \rightarrow$ is
  - $\langle NP_2 \rangle \rightarrow \langle Art \rangle \langle adj \rangle \langle N_2 \rangle$
  - $\langle Art \rangle \rightarrow$ a
  - $\langle adj \rangle \rightarrow$ beautiful
  - $\langle N_2 \rangle \rightarrow$ city

- Derivation:
  - $\langle S \rangle \Rightarrow \langle NP_1 \rangle \langle VP \rangle$
  - $\Rightarrow \langle PropN \rangle \langle VP \rangle$
  - $\Rightarrow$ **Paris** $\langle VP \rangle$
  - $\Rightarrow$ **Paris** $\langle Verb \rangle \langle NP_2 \rangle$

Grammar

- These are called rules or production which are sometimes called production rules.
- Using these rules we can derive the sentence "**Paris is a beautiful city**"
- How is the derivation done ?

- The rules are:
  - $\langle S \rangle \rightarrow \langle NP_1 \rangle \langle VP \rangle$
  - $\langle NP_1 \rangle \rightarrow \langle PropN \rangle$
  - $\langle PropN \rangle \rightarrow$ Paris
  - $\langle VP \rangle \rightarrow \langle Verb \rangle \langle NP_2 \rangle$
  - $\langle Verb \rangle \rightarrow$ is
  - $\langle NP_2 \rangle \rightarrow \langle Art \rangle \langle adj \rangle \langle N_2 \rangle$
  - $\langle Art \rangle \rightarrow$ a
  - $\langle adj \rangle \rightarrow$ beautiful
  - $\langle N_2 \rangle \rightarrow$ city

- Derivation:
  - $\langle S \rangle \Rightarrow \langle NP_1 \rangle \langle VP \rangle$
  - $\Rightarrow \langle PropN \rangle \langle VP \rangle$
  - $\Rightarrow$ **Paris** $\langle VP \rangle$
  - $\Rightarrow$ **Paris** $\langle Verb \rangle \langle NP_2 \rangle$
  - $\Rightarrow$ **Paris is** $\langle NP_2 \rangle$

Grammar

- These are called rules or production which are sometimes called production rules.
- Using these rules we can derive the sentence "**Paris is a beautiful city**"
- How is the derivation done ?

- The rules are:
  - $\langle S \rangle \rightarrow \langle NP_1 \rangle \langle VP \rangle$
  - $\langle NP_1 \rangle \rightarrow \langle PropN \rangle$
  - $\langle PropN \rangle \rightarrow$ Paris
  - $\langle VP \rangle \rightarrow \langle Verb \rangle \langle NP_2 \rangle$
  - $\langle Verb \rangle \rightarrow$ is
  - $\langle NP_2 \rangle \rightarrow \langle Art \rangle \langle adj \rangle \langle N_2 \rangle$
  - $\langle Art \rangle \rightarrow$ a
  - $\langle adj \rangle \rightarrow$ beautiful
  - $\langle N_2 \rangle \rightarrow$ city

- Derivation:
  - $\langle S \rangle \Rightarrow \langle NP_1 \rangle \langle VP \rangle$
  - $\Rightarrow \langle PropN \rangle \langle VP \rangle$
  - $\Rightarrow$ **Paris** $\langle VP \rangle$
  - $\Rightarrow$ **Paris** $\langle Verb \rangle \langle NP_2 \rangle$
  - $\Rightarrow$ **Paris is** $\langle NP_2 \rangle$
  - $\Rightarrow$ **Paris is** $\langle Art \rangle \langle adj \rangle \langle N_2 \rangle$

Grammar

- These are called rules or production which are sometimes called production rules.
- Using these rules we can derive the sentence "**Paris is a beautiful city**"
- How is the derivation done ?

- The rules are:
  - $\langle S \rangle \rightarrow \langle NP_1 \rangle \langle VP \rangle$
  - $\langle NP_1 \rangle \rightarrow \langle PropN \rangle$
  - $\langle PropN \rangle \rightarrow$ Paris
  - $\langle VP \rangle \rightarrow \langle Verb \rangle \langle NP_2 \rangle$
  - $\langle Verb \rangle \rightarrow$ is
  - $\langle NP_2 \rangle \rightarrow \langle Art \rangle \langle adj \rangle \langle N_2 \rangle$
  - $\langle Art \rangle \rightarrow$ a
  - $\langle adj \rangle \rightarrow$ beautiful
  - $\langle N_2 \rangle \rightarrow$ city

- Derivation:
  - $\langle S \rangle \Rightarrow \langle NP_1 \rangle \langle VP \rangle$
  - $\Rightarrow \langle PropN \rangle \langle VP \rangle$
  - $\Rightarrow$ **Paris** $\langle VP \rangle$
  - $\Rightarrow$ **Paris** $\langle Verb \rangle \langle NP_2 \rangle$
  - $\Rightarrow$ **Paris is** $\langle NP_2 \rangle$
  - $\Rightarrow$ **Paris is** $\langle Art \rangle \langle adj \rangle \langle N_2 \rangle$
  - $\Rightarrow$ **Paris is a** $\langle adj \rangle \langle N_2 \rangle$

Grammar

- These are called rules or production which are sometimes called production rules.
- Using these rules we can derive the sentence "**Paris is a beautiful city**"
- How is the derivation done ?

- The rules are:
  - $\langle S \rangle \rightarrow \langle NP_1 \rangle \langle VP \rangle$
  - $\langle NP_1 \rangle \rightarrow \langle PropN \rangle$
  - $\langle PropN \rangle \rightarrow$ Paris
  - $\langle VP \rangle \rightarrow \langle Verb \rangle \langle NP_2 \rangle$
  - $\langle Verb \rangle \rightarrow$ is
  - $\langle NP_2 \rangle \rightarrow \langle Art \rangle \langle adj \rangle \langle N_2 \rangle$
  - $\langle Art \rangle \rightarrow$ a
  - $\langle adj \rangle \rightarrow$ beautiful
  - $\langle N_2 \rangle \rightarrow$ city

- Derivation:
  - $\langle S \rangle \Rightarrow \langle NP_1 \rangle \langle VP \rangle$
  - $\Rightarrow \langle PropN \rangle \langle VP \rangle$
  - $\Rightarrow$ **Paris** $\langle VP \rangle$
  - $\Rightarrow$ **Paris** $\langle Verb \rangle \langle NP_2 \rangle$
  - $\Rightarrow$ **Paris is** $\langle NP_2 \rangle$
  - $\Rightarrow$ **Paris is** $\langle Art \rangle \langle adj \rangle \langle N_2 \rangle$
  - $\Rightarrow$ **Paris is a** $\langle adj \rangle \langle N_2 \rangle$
  - $\Rightarrow$ **Paris is a beautiful** $\langle N_2 \rangle$

Grammar

- These are called rules or production which are sometimes called production rules.
- Using these rules we can derive the sentence "**Paris is a beautiful city**"
- How is the derivation done ?

- The rules are:
  - $\langle S \rangle \rightarrow \langle NP_1 \rangle \langle VP \rangle$
  - $\langle NP_1 \rangle \rightarrow \langle PropN \rangle$
  - $\langle PropN \rangle \rightarrow$ Paris
  - $\langle VP \rangle \rightarrow \langle Verb \rangle \langle NP_2 \rangle$
  - $\langle Verb \rangle \rightarrow$ is
  - $\langle NP_2 \rangle \rightarrow \langle Art \rangle \langle adj \rangle \langle N_2 \rangle$
  - $\langle Art \rangle \rightarrow$ a
  - $\langle adj \rangle \rightarrow$ beautiful
  - $\langle N_2 \rangle \rightarrow$ city

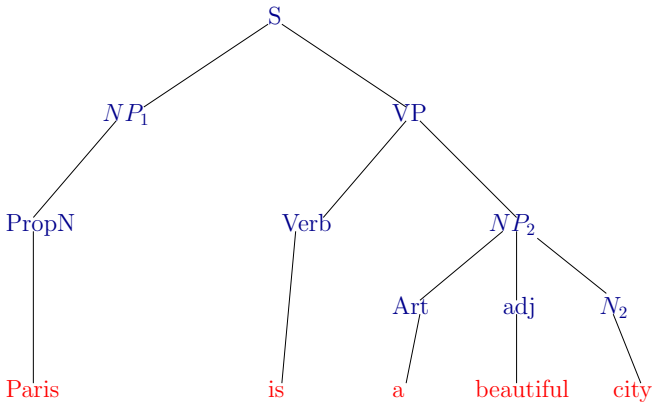
- Derivation:
  - $\langle S \rangle \Rightarrow \langle NP_1 \rangle \langle VP \rangle$
  - $\Rightarrow \langle PropN \rangle \langle VP \rangle$
  - $\Rightarrow$ **Paris** $\langle VP \rangle$
  - $\Rightarrow$ **Paris** $\langle Verb \rangle \langle NP_2 \rangle$
  - $\Rightarrow$ **Paris is** $\langle NP_2 \rangle$
  - $\Rightarrow$ **Paris is** $\langle Art \rangle \langle adj \rangle \langle N_2 \rangle$
  - $\Rightarrow$ **Paris is a** $\langle adj \rangle \langle N_2 \rangle$
  - $\Rightarrow$ **Paris is a beautiful** $\langle N_2 \rangle$
  - $\Rightarrow$ **Paris is a beautiful city**

Grammar

- The rules are:
  - $\langle S \rangle \rightarrow \langle NP_1 \rangle \langle VP \rangle$
  - $\langle NP_1 \rangle \rightarrow \langle PropN \rangle$
  - $\langle PropN \rangle \rightarrow$ Paris
  - $\langle VP \rangle \rightarrow \langle Verb \rangle \langle NP_2 \rangle$
  - $\langle Verb \rangle \rightarrow$ is
  - $\langle NP_2 \rangle \rightarrow \langle Art \rangle \langle adj \rangle \langle N_2 \rangle$
  - $\langle Art \rangle \rightarrow$ a
  - $\langle adj \rangle \rightarrow$ beautiful
  - $\langle N_2 \rangle \rightarrow$ city
- Derivation:
  - $\langle S \rangle \Rightarrow \langle NP_1 \rangle \langle VP \rangle$
  - $\Rightarrow \langle PropN \rangle \langle VP \rangle$
  - $\Rightarrow$ **Paris** $\langle VP \rangle$
  - $\Rightarrow$ **Paris** $\langle Verb \rangle \langle NP_2 \rangle$
  - $\Rightarrow$ **Paris is** $\langle NP_2 \rangle$
  - $\Rightarrow$ **Paris is** $\langle Art \rangle \langle adj \rangle \langle N_2 \rangle$
  - $\Rightarrow$ **Paris is a** $\langle adj \rangle \langle N_2 \rangle$
  - $\Rightarrow$ **Paris is a beautiful** $\langle N_2 \rangle$
  - $\Rightarrow$ **Paris is a beautiful city**

- In a rule, there is a left hand side and there is a right hand side. And, the left hand side is rewritten as the right hand side.
- '$\rightarrow$' is read as rewritten as.
- '$\Rightarrow$' represents directly derives.

Grammar



- There is a distinction between these words and the syntactic categories.
- The noun, verb phrase is rewritten as something else, so they are called '**Non-terminals**'.
- Whereas, the words 'paris', 'is', 'a', 'beautiful', 'city', they can not be rewritten as something else, because the derivation terminates there, so these are called '**Terminals**'.

Grammar

- Usually, we denotes the set of Non-Terminals by '**N**' and set of Terminals by '**T**'.
- If we consider the total alphabet V, then $V = N \cup T$
- S is the start symbol or sentence symbol, it is one of the Non-Terminals, i.e, $S \in N$.
- A grammar mainly consists of a set of Non-Terminals, a set of Terminals, a set of production rules (usually denoted by P), and a start symbol S.

Grammar

- Usually, we denotes the set of Non-Terminals by '**N**' and set of Terminals by '**T**'.
- If we consider the total alphabet V, then $V = N \cup T$
- S is the start symbol or sentence symbol, it is one of the Non-Terminals, i.e, $S \in N$.
- A grammar mainly consists of a set of Non-Terminals, a set of Terminals, a set of production rules (usually denoted by P), and a start symbol S.
- A grammar G is defined as a quadruple: G = (N, T, S, P), where
  - N is a finite set of Non-Terminals
  - T is a finite set of Terminals
  - $S \in N$ is a special symbol called the start symbol
  - P is a finite set of productions
- This is the definition of the grammar which chomsky gave.

Grammar

- Derivation:
  - $\langle S \rangle \Rightarrow \langle NP_1 \rangle \langle VP \rangle$
  - $\Rightarrow \langle PropN \rangle \langle VP \rangle$
  - $\Rightarrow$ **Paris** $\langle VP \rangle$
  - $\Rightarrow$ **Paris** $\langle Verb \rangle \langle NP_2 \rangle$
  - $\Rightarrow$ **Paris is** $\langle NP_2 \rangle$
  - $\Rightarrow$ **Paris is** $\langle Art \rangle \langle adj \rangle \langle N_2 \rangle$
  - $\Rightarrow$ **Paris is a** $\langle adj \rangle \langle N_2 \rangle$
  - $\Rightarrow$ **Paris is a beautiful** $\langle N_2 \rangle$
  - $\Rightarrow$ **Paris is a beautiful city**

- In this derivation, I have specifically taken in such a manner that I always replace the left most Non-Terminals.
- If you always replace the left most Non-Terminal, such a derivation is called a left most derivation.
- If you always replace the right most Non-Terminal, such a derivation is called a right most derivation.

Chomsky Hierarchy

- Chomsky defined 4 types of grammar.
  - Type-0
  - Type-1
  - Type-2
  - Type-3

Chomsky Hierarchy

- Chomsky defined 4 types of grammar.
  - Type-0
  - Type-1
  - Type-2
  - Type-3

- Type-0 grammar:
  - Type-0 is the most general one, some times that is called **unrestricted grammar** or **phrase structure grammar**.
  - The rules are of the form: $u \to v$, where $u \in V^*NV^*$, $v \in V^*$ and $V = N \cup T$
    - u is a string of Non-Terminals and Terminals but it should have atleast one Non-Terminals.
    - If it is fully Terminal, you can not rewrite it as something else, the derivation terminate there.

Chomsky Hierarchy

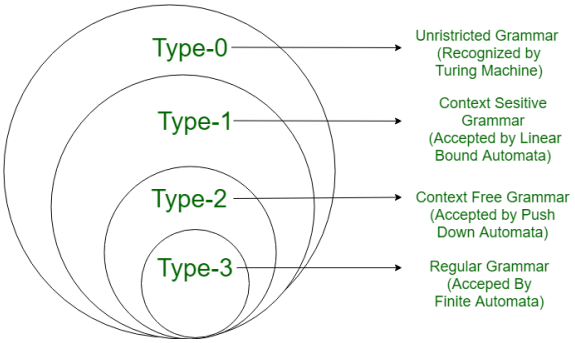
- Type-1 grammar:
  - Type-1 is also of the form $u \to v$, but the restriction is the length of the left hand side should be less than or equal to the length of the right hand side, i.e, $|u| \leq |v|$.
  - This is also called **length increasing grammar**.
  - This is also equivalent to what is known as a **context sensitive grammar**, but in context sensitive grammar the rules are of the form:
    - $\alpha A \beta \to \alpha \gamma \beta$, where $\alpha, \beta \in V^*$, $A \in N$, $\gamma \in V^+$
    - A is rewritten as γ in the context of α and β, that is why such a rule is called a context sensitive rule.
  - Generally, we take $u \to v$ where $|u| \leq |v|$. This is called type-1 grammar.
- With this restriction, we see that the empty string ϵ can not occur on the right hand side in a type-1 grammar.
  - If you want to allow ϵ and want to include the rule $S \to \epsilon$ in the language, then you make sure that start symbol S does not occur on the right hand side of any rule.

Chomsky Hierarchy

- Type-2 grammar:
  - Type-2 rules are of the form : $A \to \alpha$, where $A \in N$, $\alpha \in V^*$
    - ⋆ Left hand side you have a Non-Terminal and the right hand side you can have any string, such a rule is called **context free rule**.

- Type-3 grammar:
  - Type-3 rules are of the form : $A \to aB$, $A \to b$ where A, $B \in N$, $a \in T$ and $b \in T \cup \{\epsilon\}$
  - This is known as **regular grammar**.

Chomsky Hierarchy



- **Type-0** is the most general one, then we have given some restriction on this. We are putting the restriction on the form of production rule.
- We put the restriction on type-0, that the length of the right hand side is greater than or equal to the length of left hand side, we get **type-1**.
- Then we make restriction that the left hand side can be only a Non-Terminal, we get **type-2**.
- Then we put more restriction that, at most one Non-Terminal appears on the right side of any production, we get **type-3**.

Language generated by a Grammar

- What is the language generated by a grammar ?

Language generated by a Grammar

- What is the language generated by a grammar ?
- Let G = (N, T, S, P) be a grammar. The language generated by a grammar is denoted by L(G). Then the set

$$\mathrm{L(G)} = \{ w | w \in T^*, S \overset{*}{\Rightarrow} w \}$$

- $\overset{*}{\Rightarrow}$ is the reflexive transitive closure of $\Rightarrow$
- Starting from S, the set of Terminal strings which you can drive, define the language generated by the grammar.

Language generated by a Grammar

- What is the language generated by a grammar ?
- Let G = (N, T, S, P) be a grammar. The language generated by a grammar is denoted by L(G). Then the set

$$\text{L(G)} = \{w | w \in T^*, S \stackrel{*}{\Rightarrow} w \}$$

- $\stackrel{*}{\Rightarrow}$ is the reflexive transitive closure of $\Rightarrow$
- Starting from S, the set of Terminal strings which you can drive, define the language generated by the grammar.
- **Example:** Consider the grammar G = ($\{S\}$, {a, b}, S, P), with P given by $S \rightarrow aSb$, $S \rightarrow \epsilon$
  - Then, $S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aabb$
  - We can write, $S \stackrel{*}{\Rightarrow} aabb$
  - The string $aabb$ is a sentence in the language generated by G.

Turing Machines

Raju Hazari

Department of Computer Science and Engineering
National Institute of Technology Calicut

August 23-30, 2021

Introduction

- Introduced by Alonzo Church in 1930s. Church, along with several mathematicians at that time, was looking for the meaning of effective computability.

Introduction

- Introduced by Alonzo Church in 1930s. Church, along with several mathematicians at that time, was looking for the meaning of effective computability.
- Machines were known that were effective in computing the solutions for problems. But on a case by case basis.

Introduction

- Introduced by Alonzo Church in 1930s. Church, along with several mathematicians at that time, was looking for the meaning of effective computability.
- Machines were known that were effective in computing the solutions for problems. But on a case by case basis.
- was there a model that could capture the computability of many such machines ?

Introduction

- Introduced by Alonzo Church in 1930s. Church, along with several mathematicians at that time, was looking for the meaning of effective computability.
- Machines were known that were effective in computing the solutions for problems. But on a case by case basis.
- was there a model that could capture the computability of many such machines ?
- Turing machine is one such machine model.

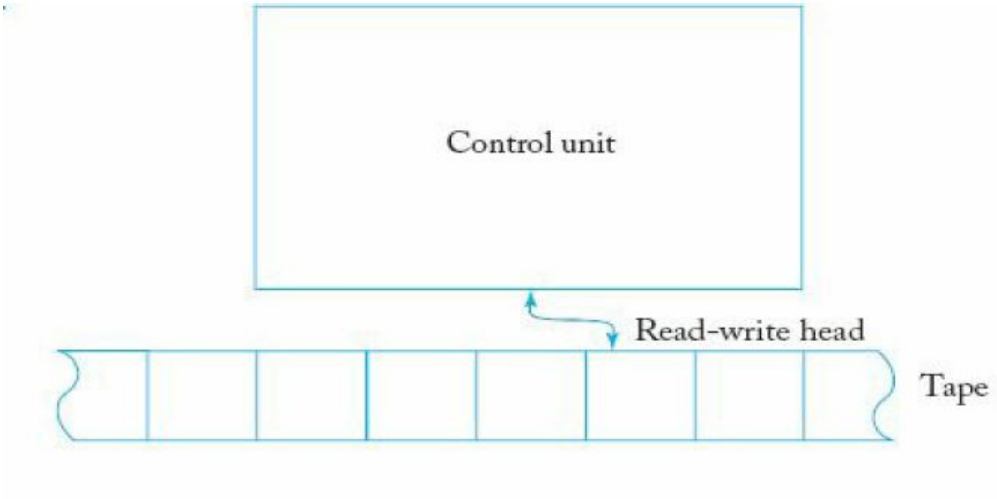
Introduction

- Introduced by Alonzo Church in 1930s. Church, along with several mathematicians at that time, was looking for the meaning of effective computability.
- Machines were known that were effective in computing the solutions for problems. But on a case by case basis.
- was there a model that could capture the computability of many such machines ?
- Turing machine is one such machine model.
- The machine model was defined by Alan Turing in 1936 and he called the basic model as a "computer".

Introduction

- Introduced by Alonzo Church in 1930s. Church, along with several mathematicians at that time, was looking for the meaning of effective computability.
- Machines were known that were effective in computing the solutions for problems. But on a case by case basis.
- was there a model that could capture the computability of many such machines ?
- Turing machine is one such machine model.
- The machine model was defined by Alan Turing in 1936 and he called the basic model as a "computer".
- He stated "What could naturally be called an effective procedure can be realised by a Turing machine".
  –This is known as **Church-Turing hypothesis**.

Turing Machines



An intuitive visualization of a Turing machine

Turing Machines

- The machine consists of a **finite control**, which can be in any of a finite set of states.

Turing Machines

- The machine consists of a **finite control**, which can be in any of a finite set of states.
- There is a **tape** divided into cells; each cell can hold any one of a finite number of symbols.
  - The input, which is a finite-length string of symbols chosen from the input alphabet, is placed on the tape.
  - All other tape cells, extending infinitely to the left and right, initially holds a special symbol called the *blank*.
  - The blank is a tape symbol, but not an input symbol.

Turing Machines

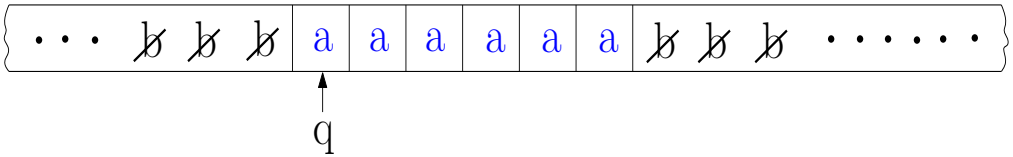
- The machine consists of a **finite control**, which can be in any of a finite set of states.
- There is a **tape** divided into cells; each cell can hold any one of a finite number of symbols.
  - The input, which is a finite-length string of symbols chosen from the input alphabet, is placed on the tape.
  - All other tape cells, extending infinitely to the left and right, initially holds a special symbol called the *blank*.
  - The blank is a tape symbol, but not an input symbol.
- There is a tape head (**read-write head**) that is always positioned at one of the tape cells.
  - Initially, the tape head is at the leftmost cell that holds the input.
  - The read-write head can move right or left on the tape and that can read and write a single symbol on each move.

Turing Machines

- A *move* of the Turing machine is a function of the state of the finite control and tape symbol scanned.

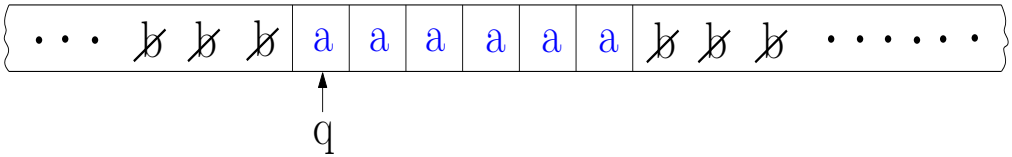
Turing Machines

- A *move* of the Turing machine is a function of the state of the finite control and tape symbol scanned.

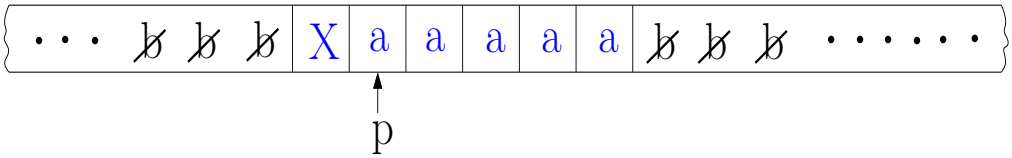


Turing Machines

- A *move* of the Turing machine is a function of the state of the finite control and tape symbol scanned.

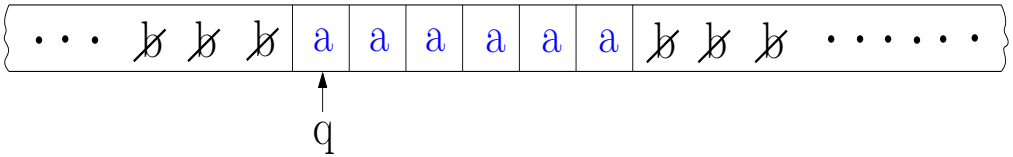


$$\textbf{\textit{Move}}: \delta(q, a) = (p, X, R)$$

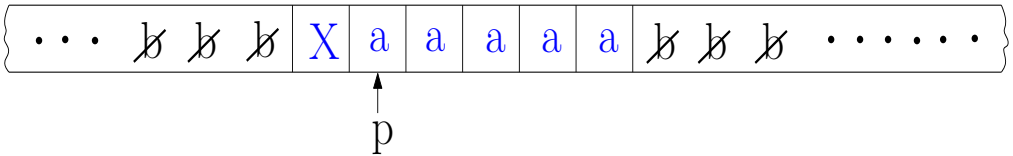


Turing Machines

- A *move* of the Turing machine is a function of the state of the finite control and tape symbol scanned.



$$\textbf{\textit{Move}} : \delta(q, a) = (p, X, R)$$



- In one move, the Turing machine will :
  1. *Change state.* The next state optionally may be the same as the current state.
  2. *Write a tape symbol in the cell scanned.* The symbol written may be the same as the symbol currently there.
  3. *Move the tape head left or right.*

Turing Machines

- We formally define a Turing Machine (TM) by the 7-tuple :

$$\mathrm{M} = (\mathrm{Q},\ \Sigma,\ \Gamma,\ \delta,\ q_0,\ \not\emptyset,\ \mathrm{F})$$

where

  - Q : The finite set of states of the finite control
  - Σ : The finite set of input symbols
  - Γ : The complete set of tape symbols; Σ is always a subset of Γ
  - δ : The transition function. The transition function δ is defined as

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

  - q_0 : The start state, a member of Q ($q_0 \in Q$)
  - $\not\emptyset$: The blank symbol. This symbol is in Γ but not in Σ
  - F : The set of final or accepting states, a subset of Q ($F \subseteq Q$)

Turing Machines

What is the difference between finite automata and Turing machines ?

Turing Machines

What is the difference between finite automata and Turing machines ?

- A Turing machine can both write on the tape and read from it.

Turing Machines

What is the difference between finite automata and Turing machines ?

- A Turing machine can both write on the tape and read from it.
- The read-write head can move both to the left and to the right.

Turing Machines

What is the difference between finite automata and Turing machines ?

- A Turing machine can both write on the tape and read from it.
- The read-write head can move both to the left and to the right.
- The tape is infinite.

Turing Machines (A input output device)

- We can look a Turing machine as an **input output device** or an **accepting device**.

Turing Machines (A input output device)

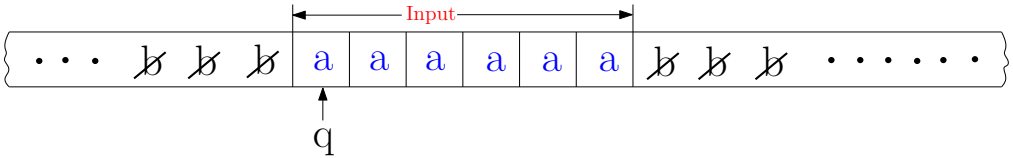
- We can look a Turing machine as an **input output device** or an **accepting device**.
- When we look at it as an *accepting device*, it accepts type 0 languages.

Turing Machines (A input output device)

- We can look a Turing machine as an **input output device** or an **accepting device**.
- When we look at it as an *accepting device*, it accepts type 0 languages.
- When we look at it as a *input output device*

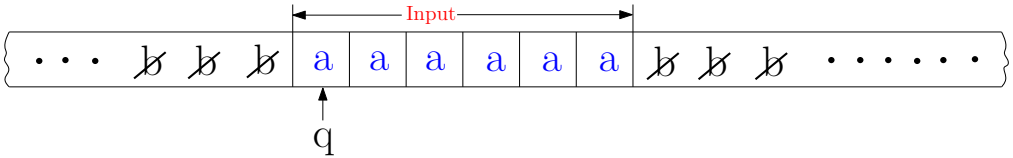
Turing Machines (A input output device)

- We can look a Turing machine as an **input output device** or an **accepting device**.
- When we look at it as an *accepting device*, it accepts type 0 languages.
- When we look at it as a *input output device*
  - The **input** is the non-blank portion when we start the machine

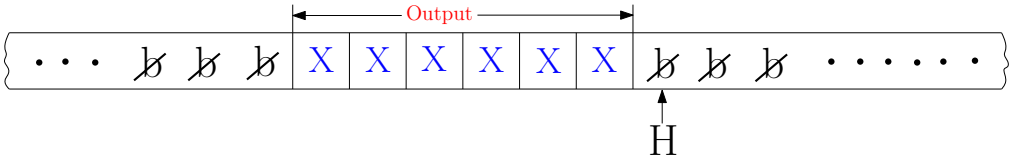


Turing Machines (A input output device)

- We can look a Turing machine as an **input output device** or an **accepting device**.
- When we look at it as an *accepting device*, it accepts type 0 languages.
- When we look at it as a *input output device*
  - The **input** is the non-blank portion when we start the machine



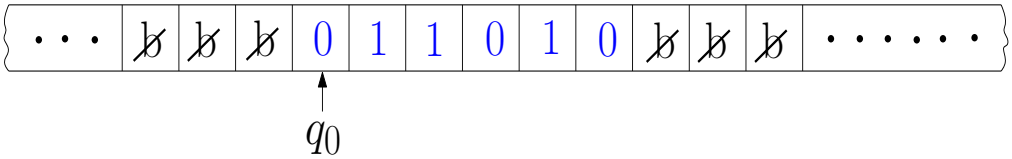
  - When the machine halts, whatever is a non-blank portion, that is the **output**.



Mapping : $\delta(q, a) = (q, X, R)$ and $\delta(q, \not b) = (\text{H}, \not b, \text{-})$

Turing Machines (A input output device)

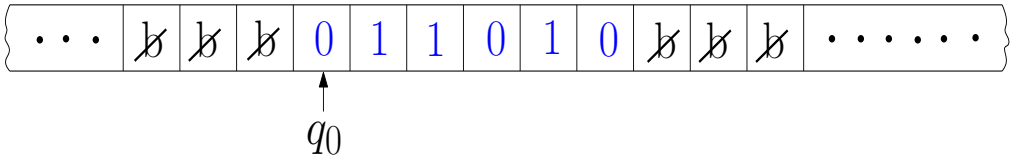
Example 1 : Parity Checker



print "**O**" if the number of 1's is odd
print "**E**" if the number of 1's is even

Turing Machines (A input output device)

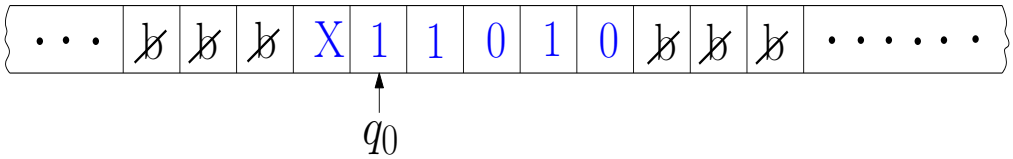
Example 1 : Parity Checker



print "**O**" if the number of 1's is odd
print "**E**" if the number of 1's is even

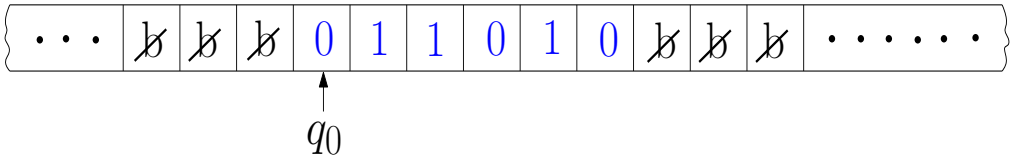
Solution :

$$Move : \delta(q_0, 0) = (q_0, X, R)$$



Turing Machines (A input output device)

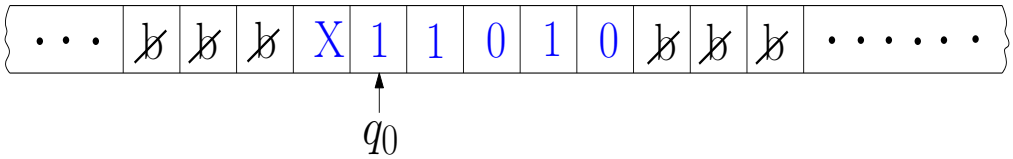
Example 1 : Parity Checker



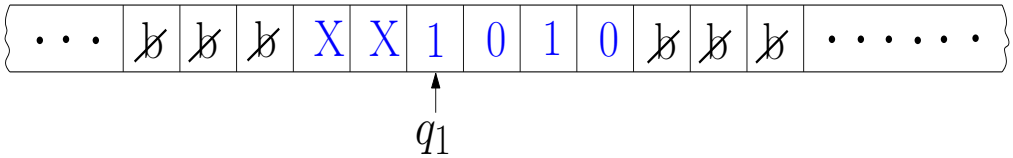
print "**O**" if the number of 1's is odd
print "**E**" if the number of 1's is even

Solution :

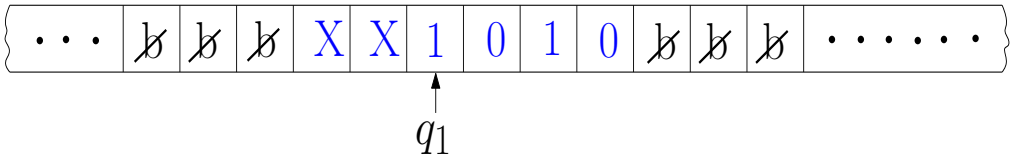
$$Move : \delta(q_0, 0) = (q_0, X, R)$$



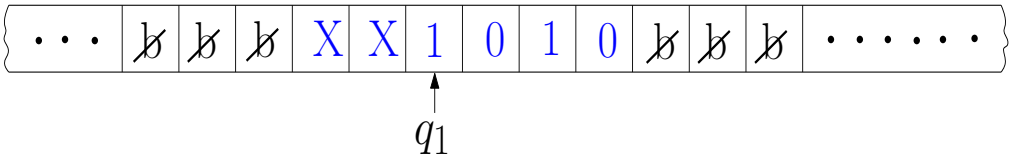
$$Move : \delta(q_0, 1) = (q_1, X, R)$$



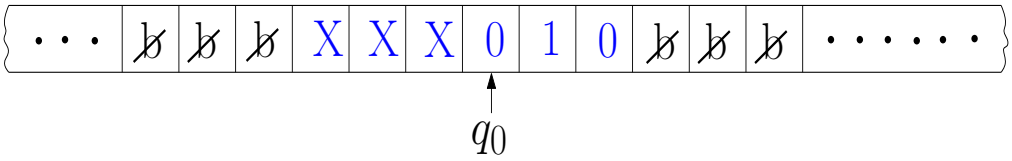
Turing Machines (A input output device)



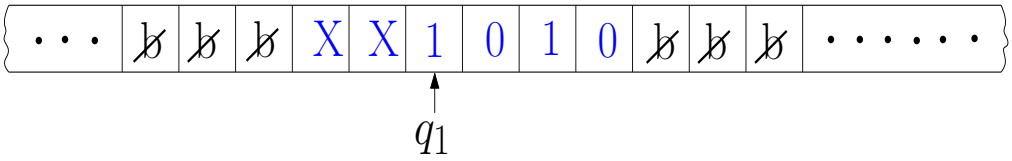
Turing Machines (A input output device)



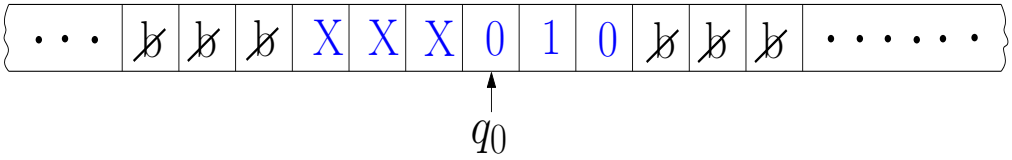
$$Move : \delta(q_1, 1) = (q_0, X, R)$$



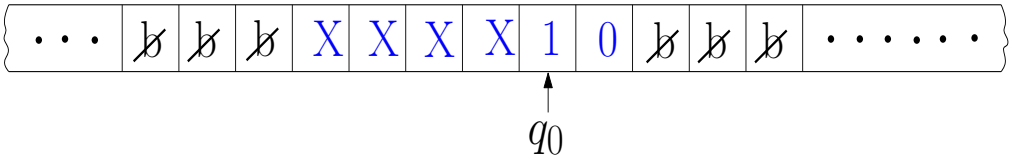
Turing Machines (A input output device)



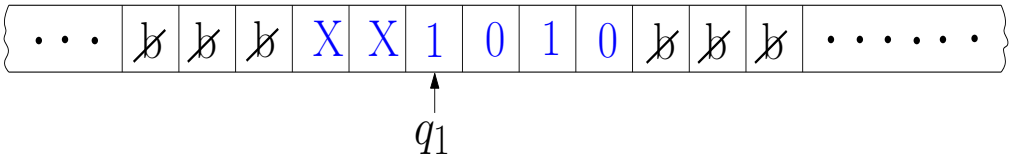
$$Move: \delta(q_1, 1) = (q_0, X, R)$$



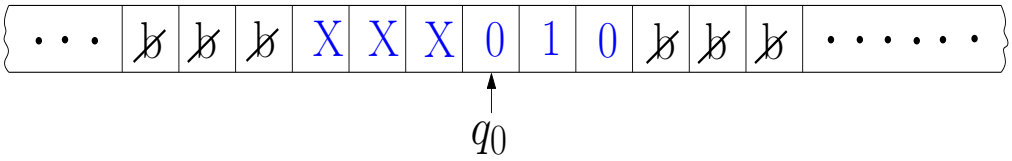
$$Move: \delta(q_0, 0) = (q_0, X, R)$$



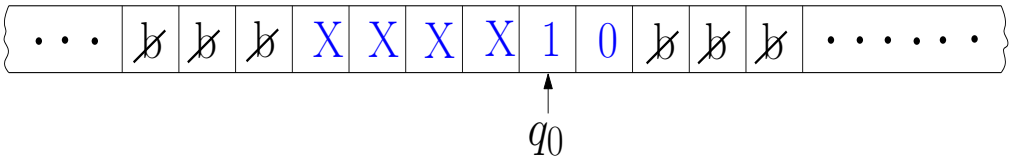
Turing Machines (A input output device)



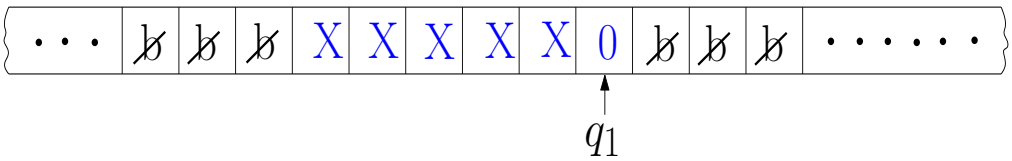
$$Move: \delta(q_1, 1) = (q_0, X, R)$$



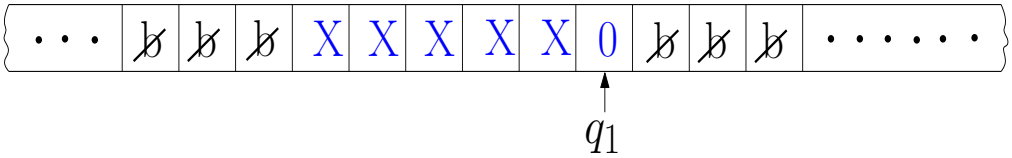
$$Move: \delta(q_0, 0) = (q_0, X, R)$$



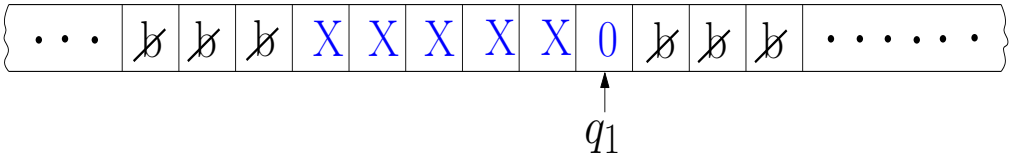
$$Move: \delta(q_0, 1) = (q_1, X, R)$$



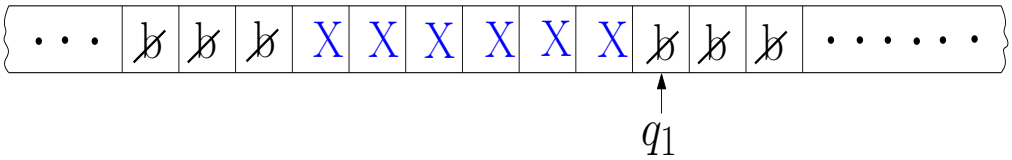
Turing Machines (A input output device)



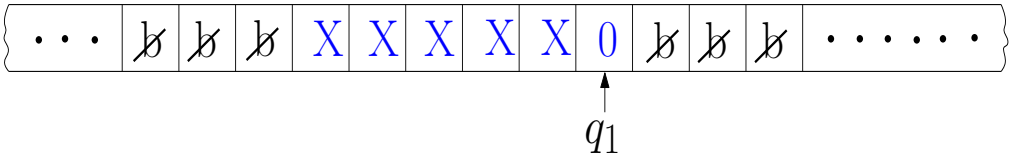
Turing Machines (A input output device)



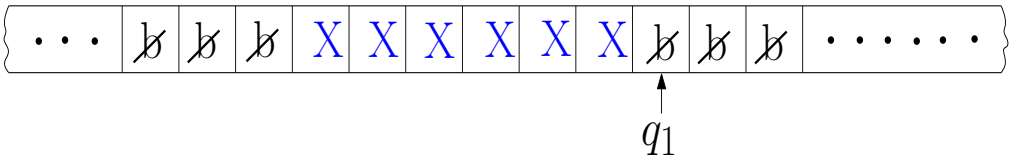
$$Move: \delta(q_1, 0) = (q_1, X, R)$$



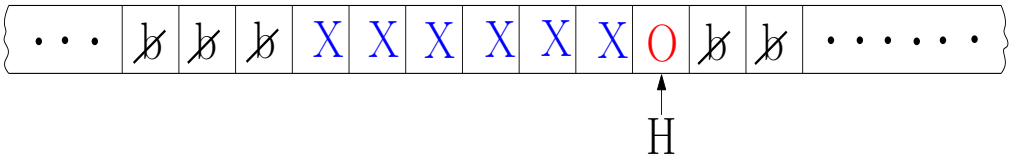
Turing Machines (A input output device)



$$Move: \delta(q_1, 0) = (q_1, X, R)$$



$$Move: \delta(q_1, \not b) = (\text{H}, \text{O}, \text{-})$$



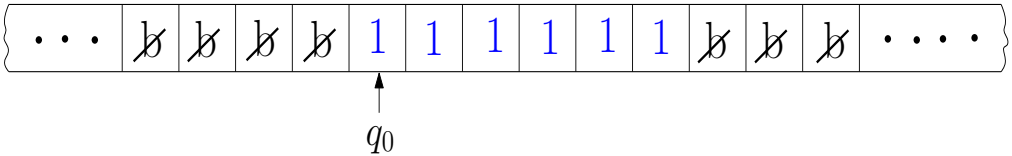
Turing Machines (A input output device)

- So, the Turing machine M = (Q, Σ, Γ, δ, q_0, ∦, F) where
  - States $Q = \{q_0, q_1\}$
  - Input alphabet $\Sigma = \{0, 1\}$
  - Tape alphabet $\Gamma = \{0, 1, ∦, X\}$
  - Mapping δ :

$$\delta(q_0, 0) = (q_0, X, R)$$
$$\delta(q_0, 1) = (q_1, X, R)$$
$$\delta(q_1, 0) = (q_1, X, R)$$
$$\delta(q_1, 1) = (q_0, X, R)$$
$$\delta(q_1, ∦) = (\text{H, O, -})$$
$$\delta(q_0, ∦) = (\text{H, E, -})$$

Turing Machines (A input output device)

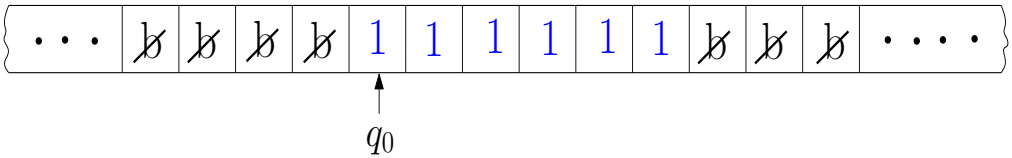
Example 2 : Unary to binary converter



print "**B B A**", where $B \to 1$ and $A \to 0$

Turing Machines (A input output device)

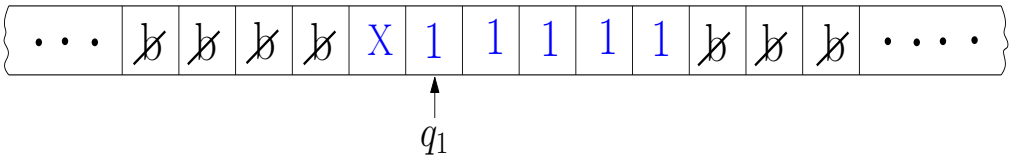
Example 2 : Unary to binary converter



print "**B B A**", where $B \to 1$ and $A \to 0$

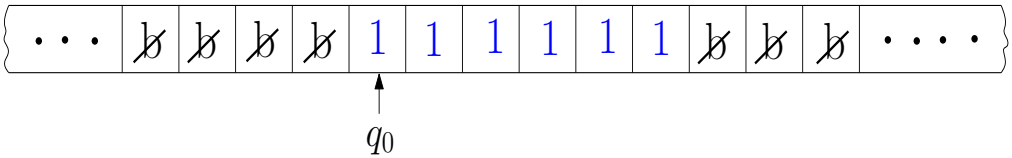
Solution :

$$Move : \delta(q_0, 1) = (q_1, X, R)$$



Turing Machines (A input output device)

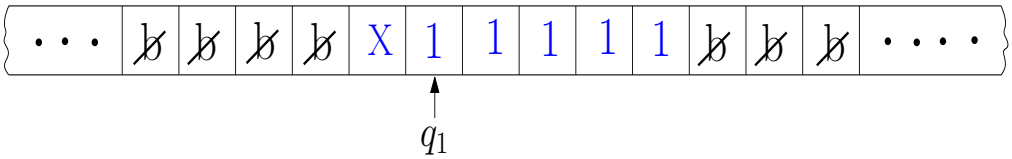
Example 2 : Unary to binary converter



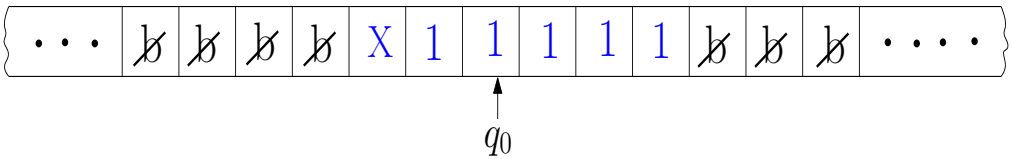
print "**B B A**", where $B \to 1$ and $A \to 0$

Solution :

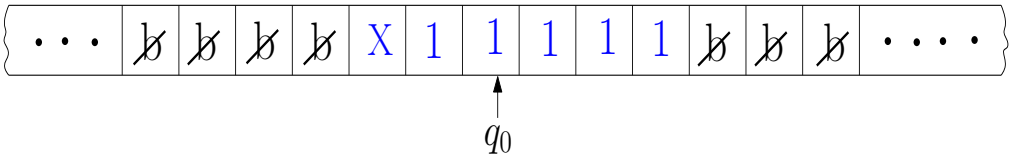
$$Move : \delta(q_0, 1) = (q_1, X, R)$$



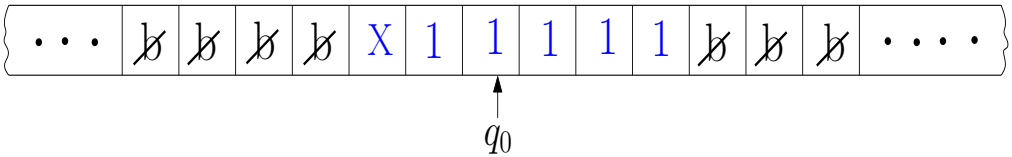
$$Move : \delta(q_1, 1) = (q_0, 1, R)$$



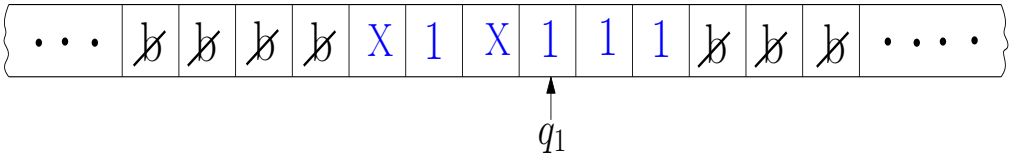
Turing Machines (A input output device)



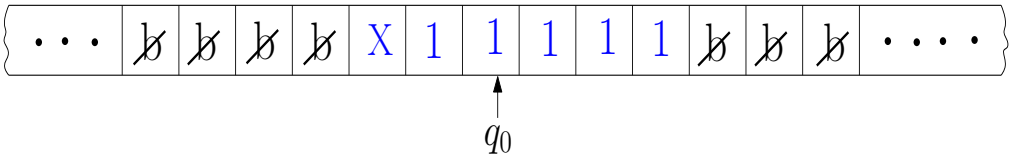
Turing Machines (A input output device)



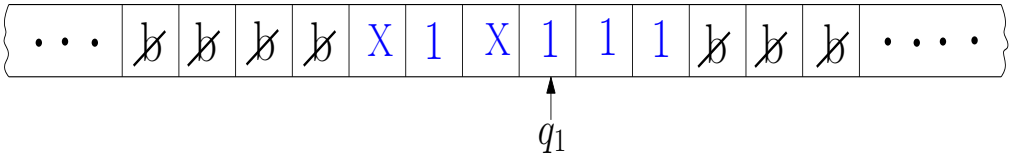
$$Move: \delta(q_0, 1) = (q_1, X, R)$$



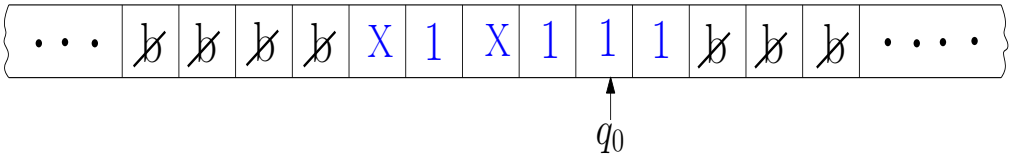
Turing Machines (A input output device)



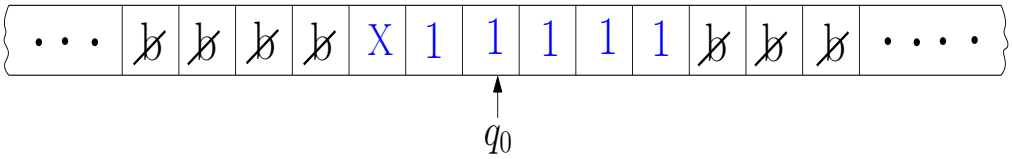
$$Move: \delta(q_0, 1) = (q_1, X, R)$$



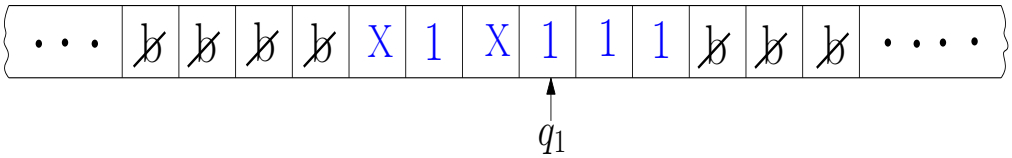
$$Move: \delta(q_1, 1) = (q_0, 1, R)$$



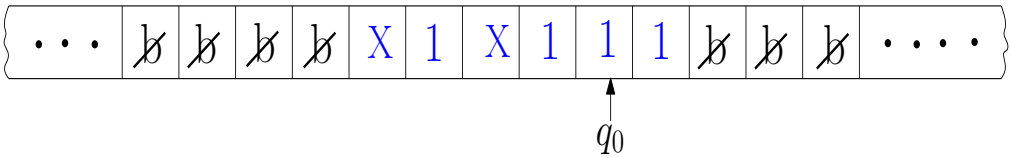
Turing Machines (A input output device)



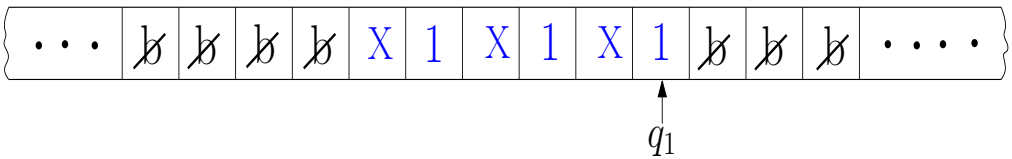
$$Move: \delta(q_0, 1) = (q_1, X, R)$$



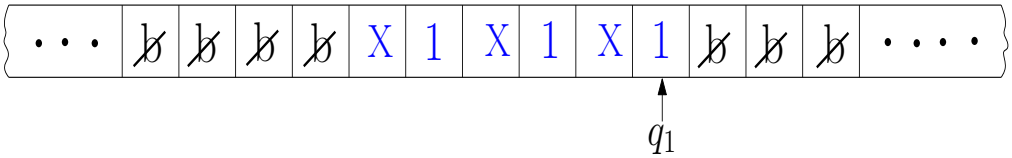
$$Move: \delta(q_1, 1) = (q_0, 1, R)$$



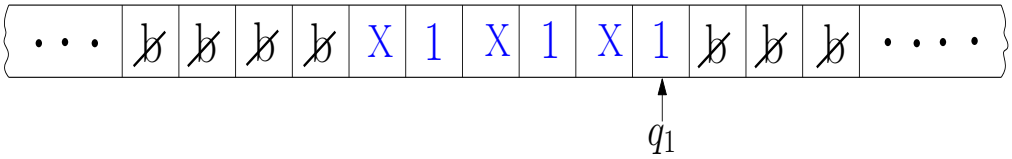
$$Move: \delta(q_0, 1) = (q_1, X, R)$$



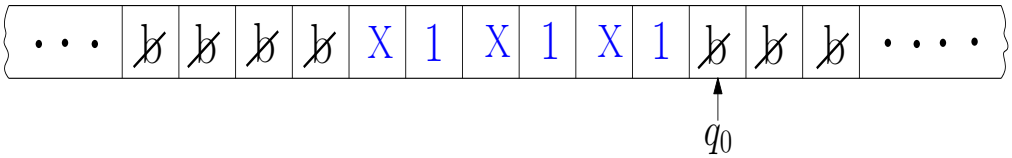
Turing Machines (A input output device)



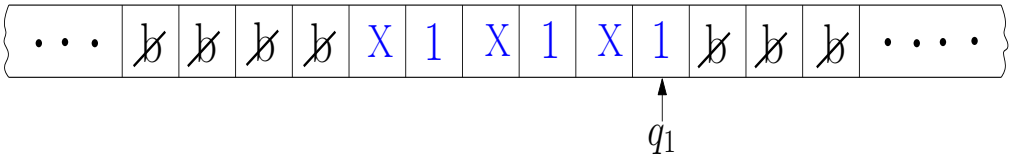
Turing Machines (A input output device)



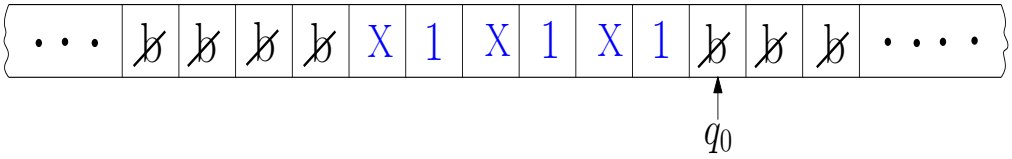
$$Move: \delta(q_1, 1) = (q_0, 1, R)$$



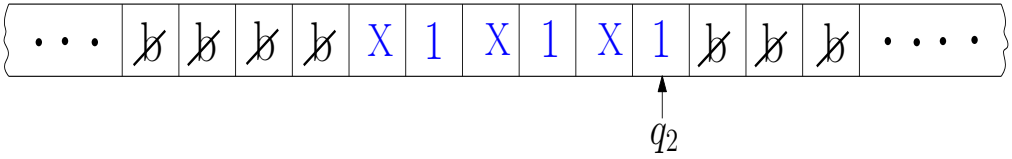
Turing Machines (A input output device)



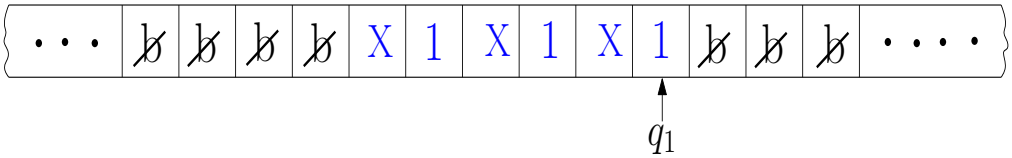
$$Move: \delta(q_1, 1) = (q_0, 1, R)$$



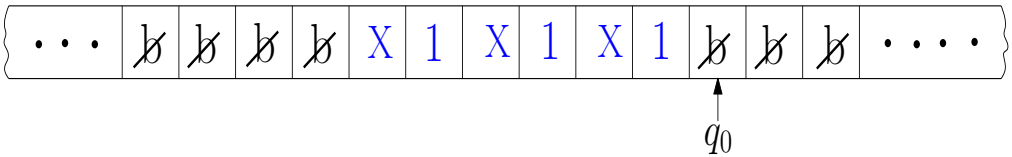
$$Move: \delta(q_0, \not b) = (q_2, \not b, \mathrm{L})$$



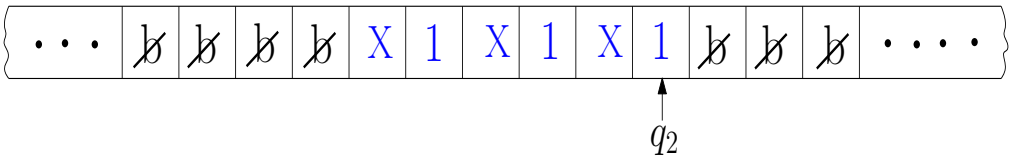
Turing Machines (A input output device)



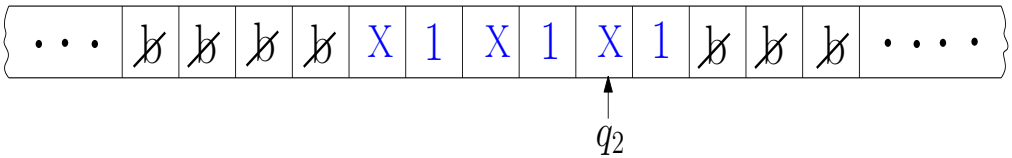
$$Move: \delta(q_1, 1) = (q_0, 1, R)$$



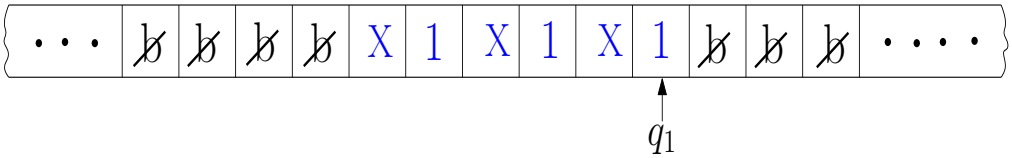
$$Move: \delta(q_0, \not\vdash) = (q_2, \not\vdash, \mathrm{L})$$



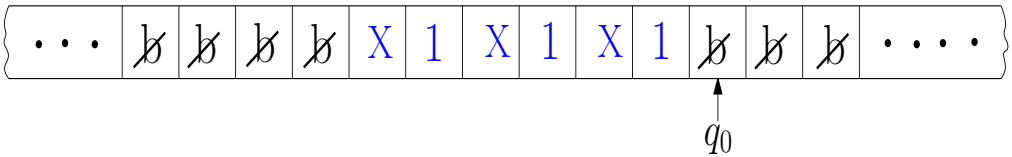
$$Move: \delta(q_2, Z) = (q_2, Z, L) \text{ where } Z \in \{\mathrm{X}, 1, \mathrm{A}, \mathrm{B}\}$$



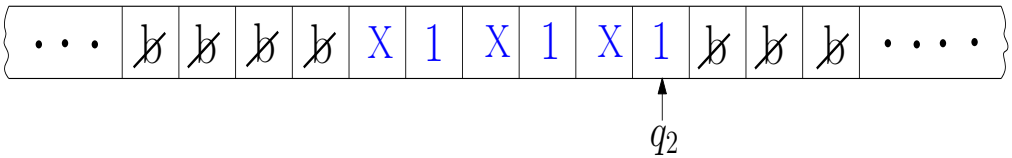
Turing Machines (A input output device)



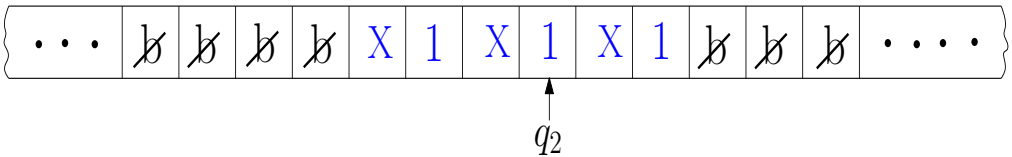
$$Move: \delta(q_1, 1) = (q_0, 1, R)$$



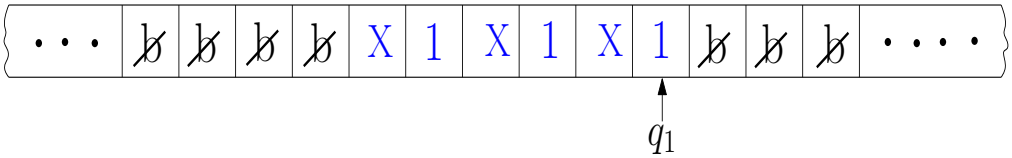
$$Move: \delta(q_0, \not\vdash) = (q_2, \not\vdash, \mathrm{L})$$



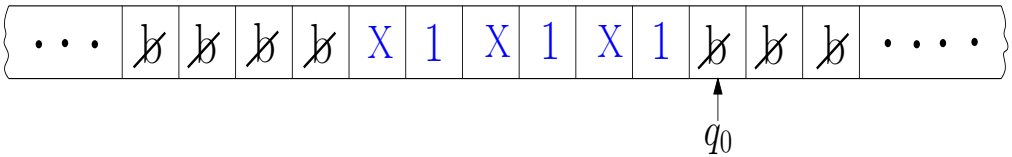
$$Move: \delta(q_2, Z) = (q_2, Z, L) \text{ where } Z \in \{\mathrm{X}, 1, \mathrm{A}, \mathrm{B}\}$$



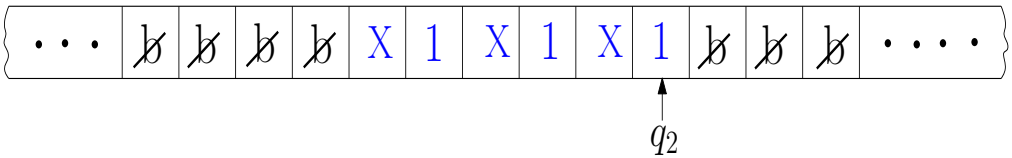
Turing Machines (A input output device)



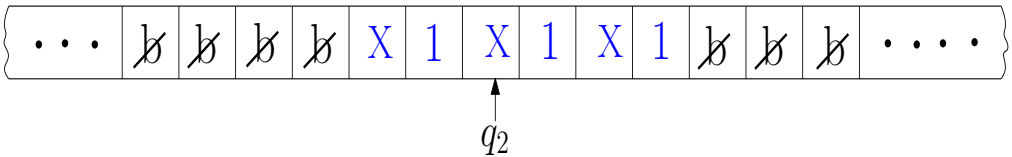
$$Move: \delta(q_1, 1) = (q_0, 1, R)$$



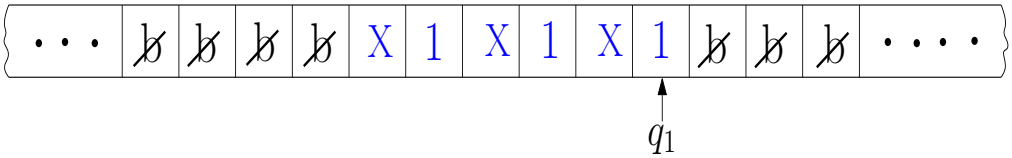
$$Move: \delta(q_0, \not\vdash) = (q_2, \not\vdash, \mathrm{L})$$



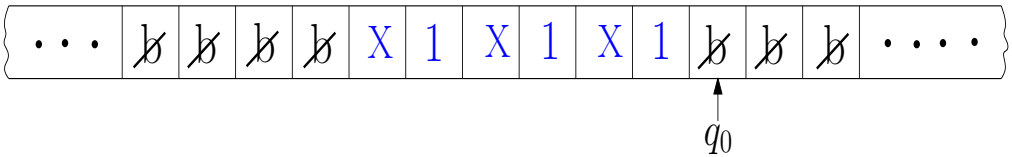
$$Move: \delta(q_2, Z) = (q_2, Z, L) \text{ where } Z \in \{\mathrm{X}, 1, \mathrm{A}, \mathrm{B}\}$$



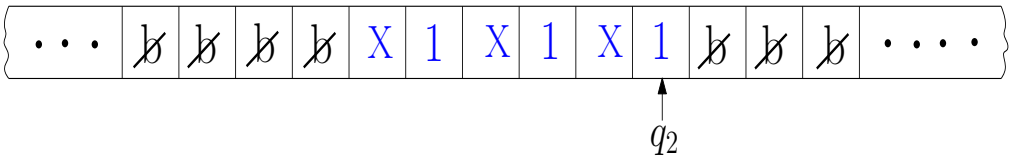
Turing Machines (A input output device)



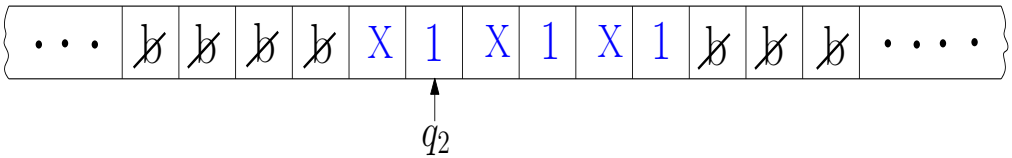
$$Move: \delta(q_1, 1) = (q_0, 1, R)$$



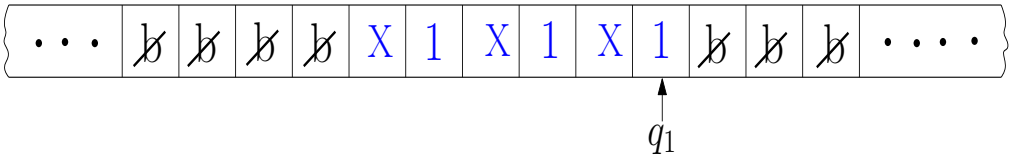
$$Move: \delta(q_0, \not\vdash) = (q_2, \not\vdash, \mathrm{L})$$



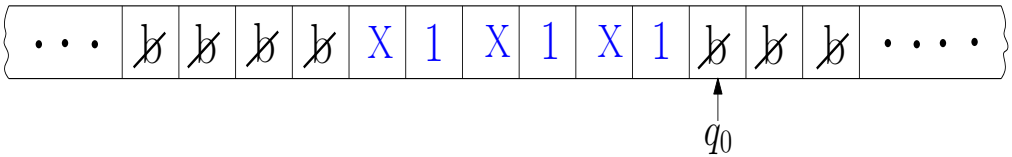
$$Move: \delta(q_2, Z) = (q_2, Z, L) \text{ where } Z \in \{\mathrm{X}, 1, \mathrm{A}, \mathrm{B}\}$$



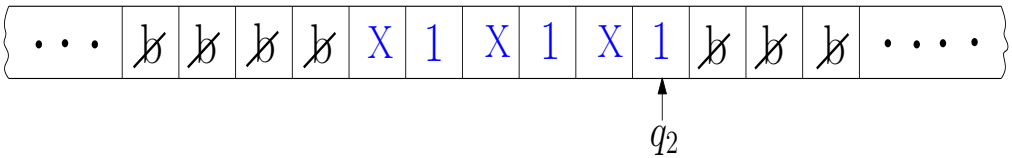
Turing Machines (A input output device)



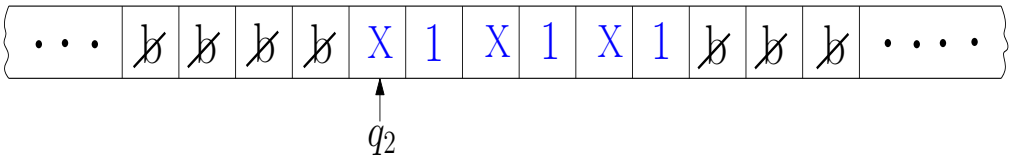
$$Move: \delta(q_1, 1) = (q_0, 1, R)$$



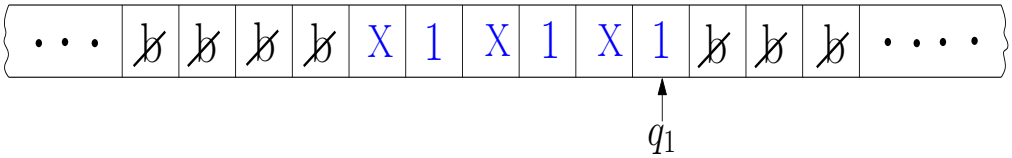
$$Move: \delta(q_0, \not b) = (q_2, \not b, \mathrm{L})$$



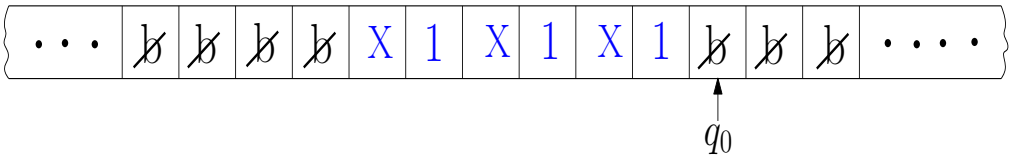
$$Move: \delta(q_2, Z) = (q_2, Z, L) \text{ where } Z \in \{\mathrm{X}, 1, \mathrm{A}, \mathrm{B}\}$$



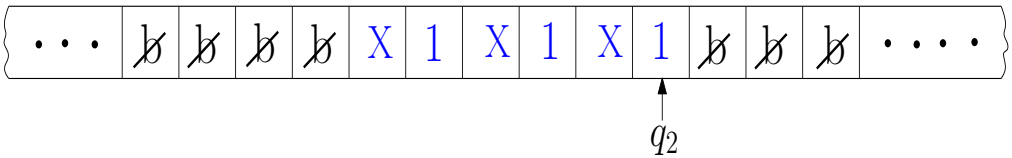
Turing Machines (A input output device)



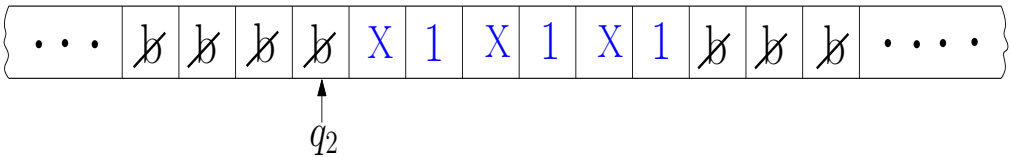
$$Move: \delta(q_1, 1) = (q_0, 1, R)$$



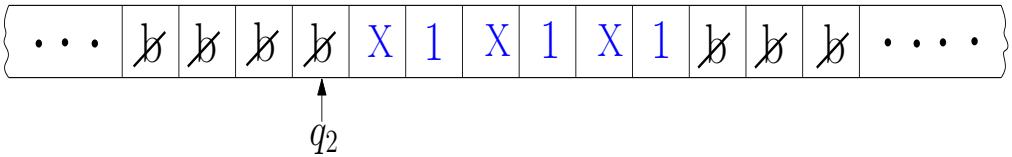
$$Move: \delta(q_0, \not b) = (q_2, \not b, \mathrm{L})$$



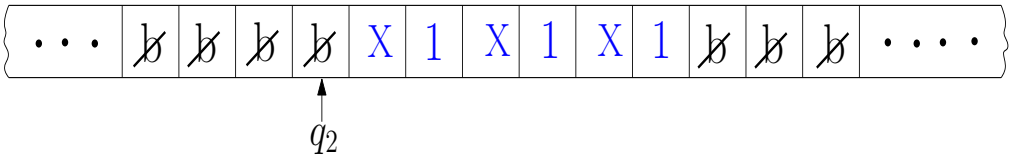
$$Move: \delta(q_2, Z) = (q_2, Z, L) \text{ where } Z \in \{\mathrm{X}, 1, \mathrm{A}, \mathrm{B}\}$$



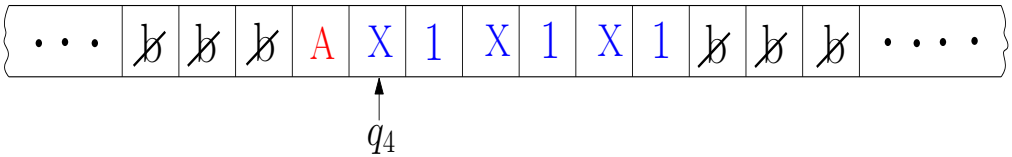
Turing Machines (A input output device)



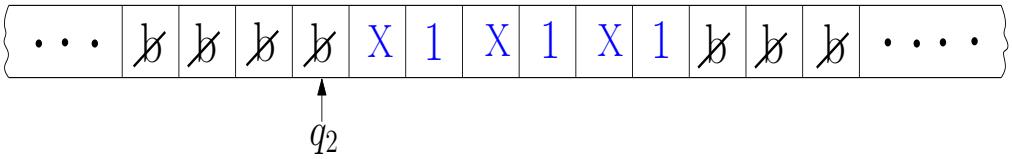
Turing Machines (A input output device)



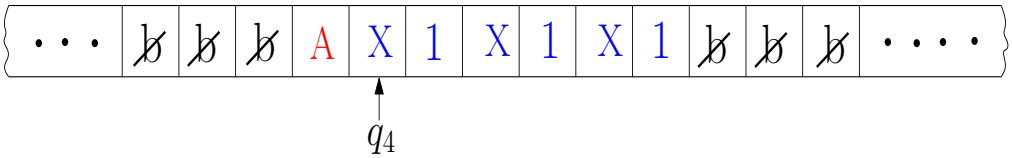
$$Move : \delta(q_2, \not\flat) = (q_4, A, R)$$



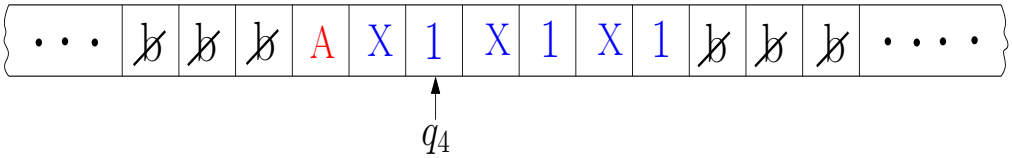
Turing Machines (A input output device)



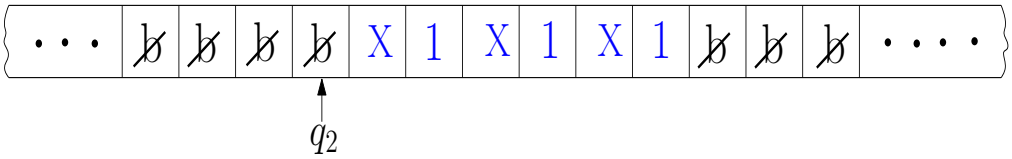
$$Move: \delta(q_2, \not b) = (q_4, A, R)$$



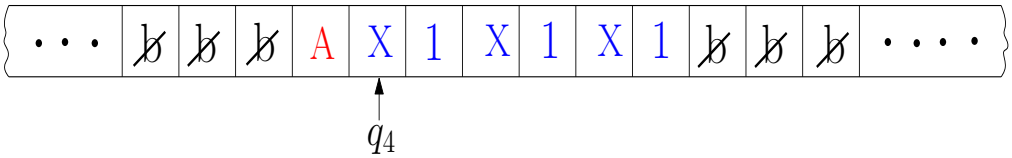
$$Move: \delta(q_4, X) = (q_4, X, R)$$



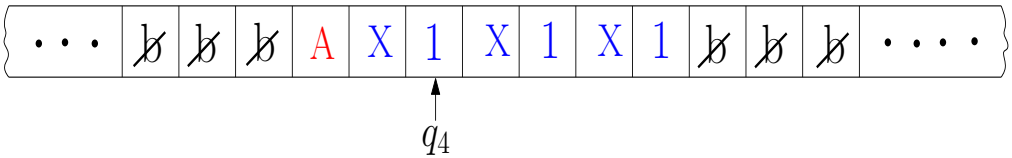
Turing Machines (A input output device)



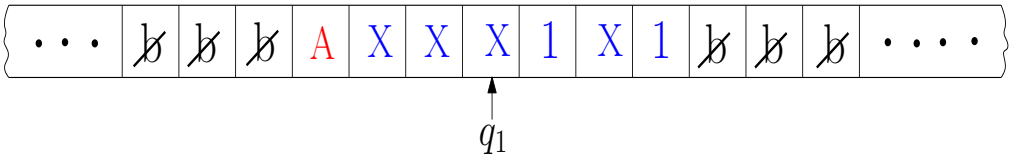
$$Move: \delta(q_2, \not b) = (q_4, A, R)$$



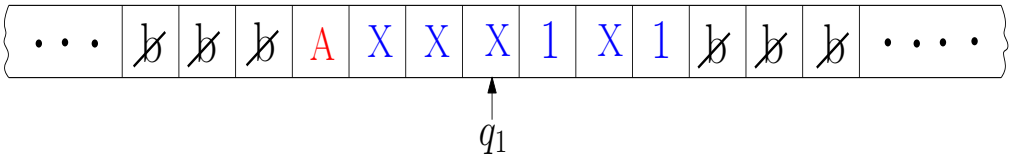
$$Move: \delta(q_4, X) = (q_4, X, R)$$



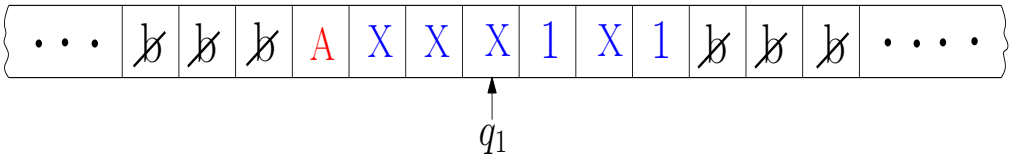
$$Move: \delta(q_4, 1) = (q_1, X, R)$$



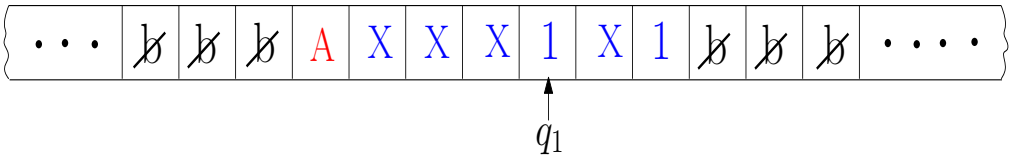
Turing Machines (A input output device)



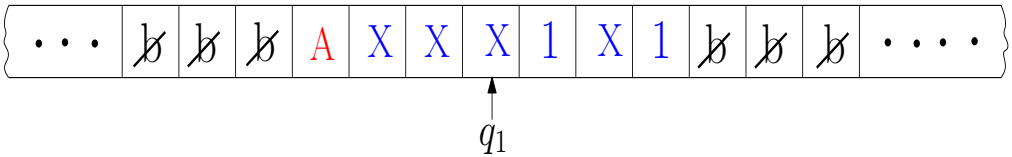
Turing Machines (A input output device)



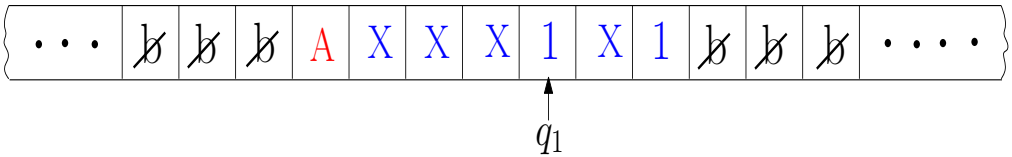
$$Move: \delta(q_1, X) = (q_1, X, R)$$



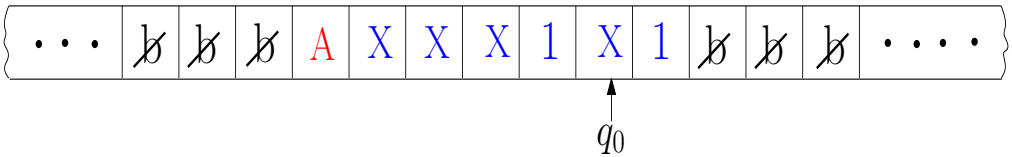
Turing Machines (A input output device)



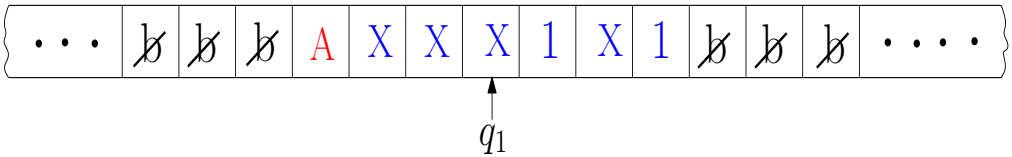
$$Move: \delta(q_1, X) = (q_1, X, R)$$



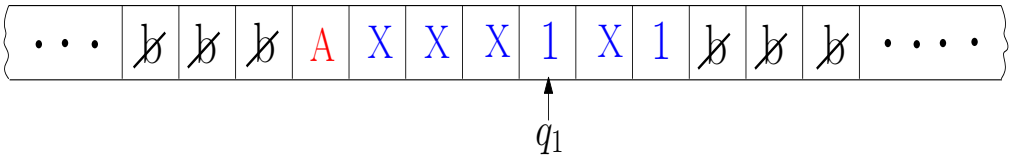
$$Move: \delta(q_1, 1) = (q_0, 1, R)$$



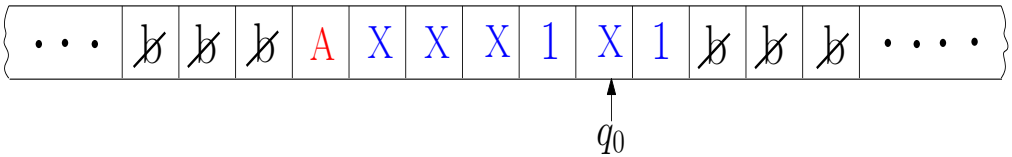
Turing Machines (A input output device)



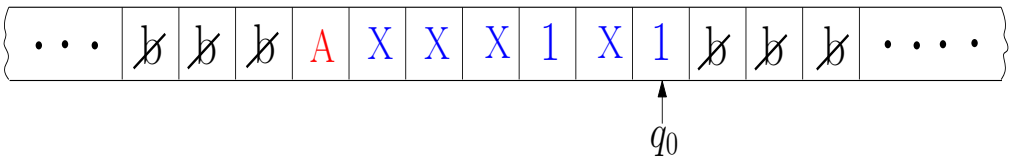
$$Move: \delta(q_1, X) = (q_1, X, R)$$



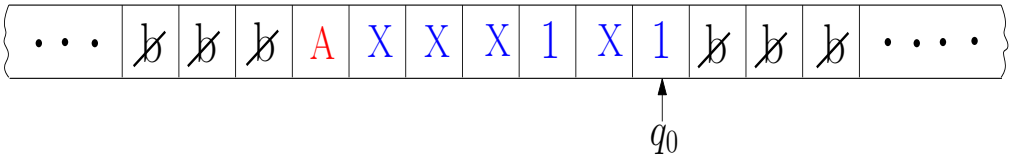
$$Move: \delta(q_1, 1) = (q_0, 1, R)$$



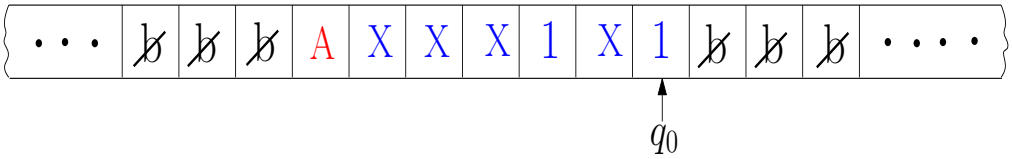
$$Move: \delta(q_0, X) = (q_0, X, R)$$



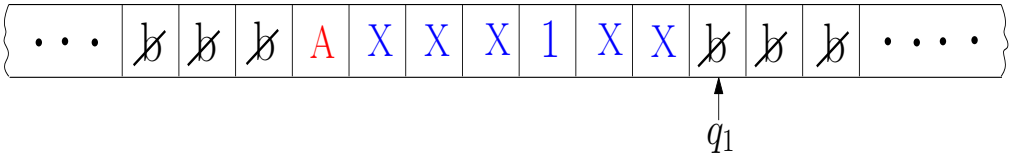
Turing Machines (A input output device)



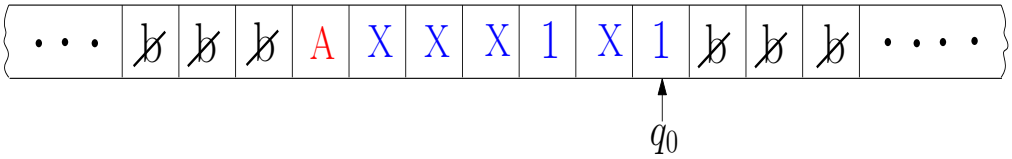
Turing Machines (A input output device)



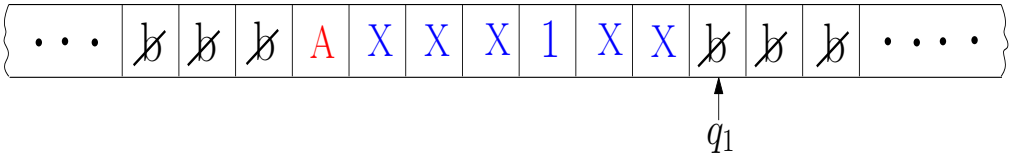
$$Move: \delta(q_0, 1) = (q_1, X, R)$$



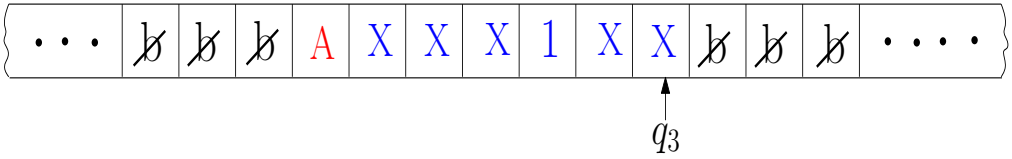
Turing Machines (A input output device)



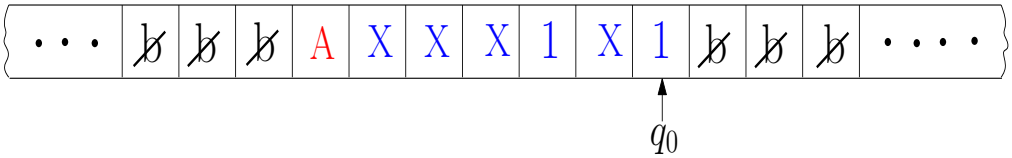
$$Move: \delta(q_0, 1) = (q_1, X, R)$$



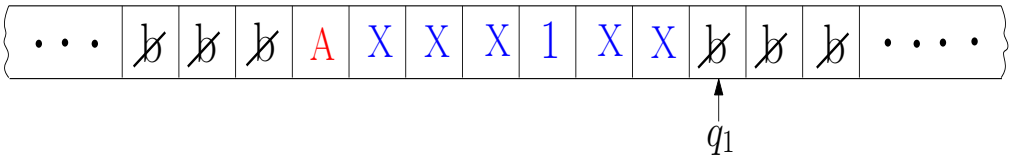
$$Move: \delta(q_1, \not b) = (q_3, \not b, \mathrm{L})$$



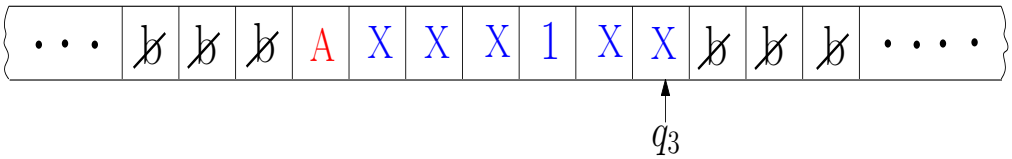
Turing Machines (A input output device)



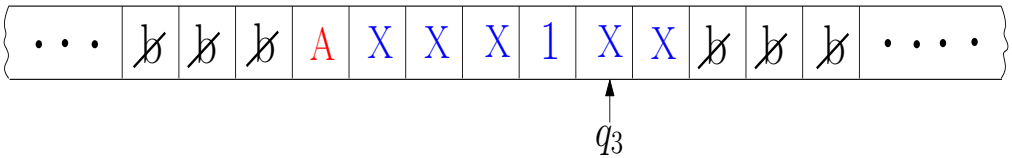
$$Move: \delta(q_0, 1) = (q_1, X, R)$$



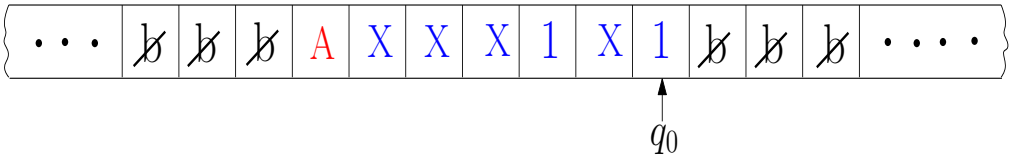
$$Move: \delta(q_1, \not\vdash) = (q_3, \not\vdash, \mathrm{L})$$



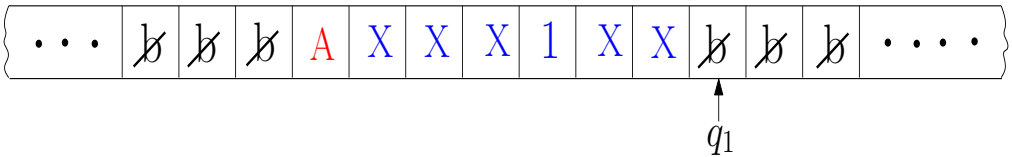
$$Move: \delta(q_3, Z) = (q_3, Z, L) \text{ where } Z \in \{\mathrm{X}, 1, \mathrm{A}, \mathrm{B}\}$$



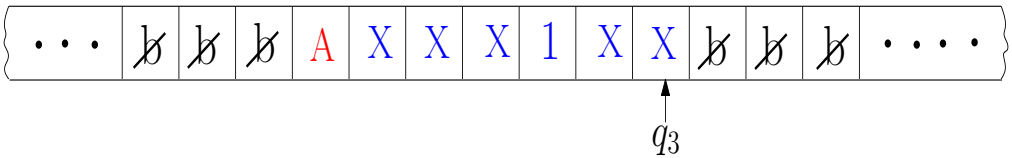
Turing Machines (A input output device)



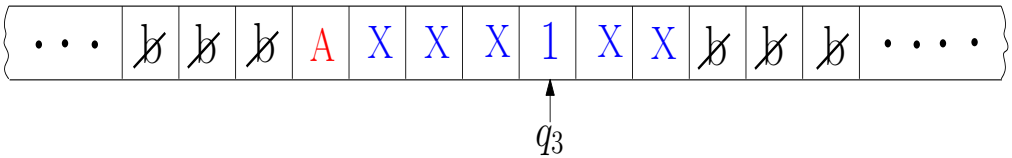
$$Move: \delta(q_0, 1) = (q_1, X, R)$$



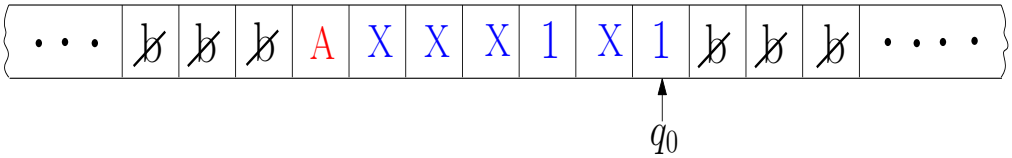
$$Move: \delta(q_1, \not\vdash) = (q_3, \not\vdash, \mathrm{L})$$



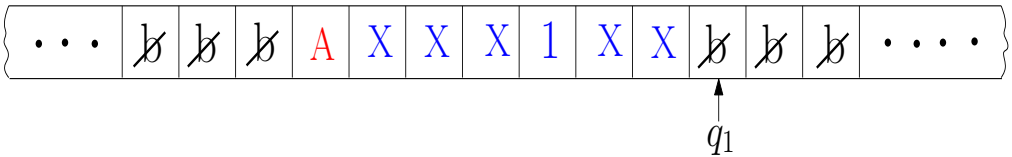
$$Move: \delta(q_3, Z) = (q_3, Z, L) \text{ where } Z \in \{\mathrm{X}, 1, \mathrm{A}, \mathrm{B}\}$$



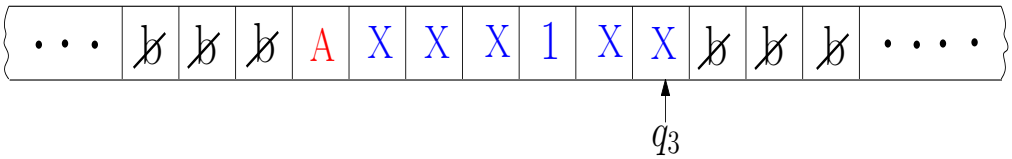
Turing Machines (A input output device)



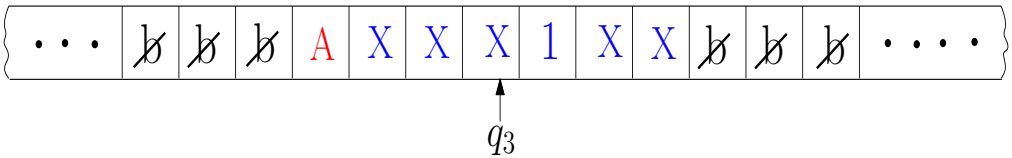
$$Move: \delta(q_0, 1) = (q_1, X, R)$$



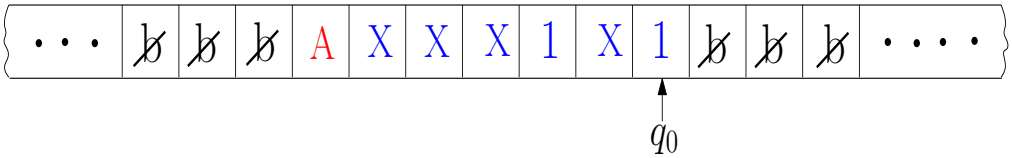
$$Move: \delta(q_1, \not\vdash) = (q_3, \not\vdash, \mathrm{L})$$



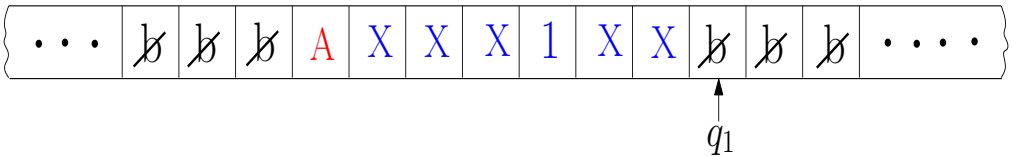
$$Move: \delta(q_3, Z) = (q_3, Z, L) \text{ where } Z \in \{\mathrm{X}, 1, \mathrm{A}, \mathrm{B}\}$$



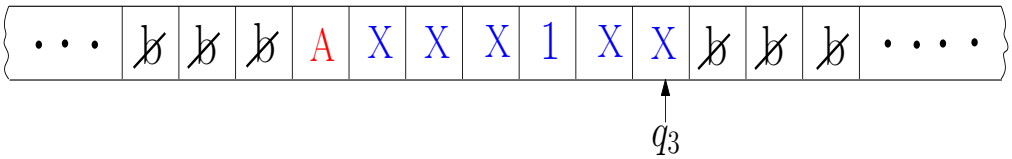
Turing Machines (A input output device)



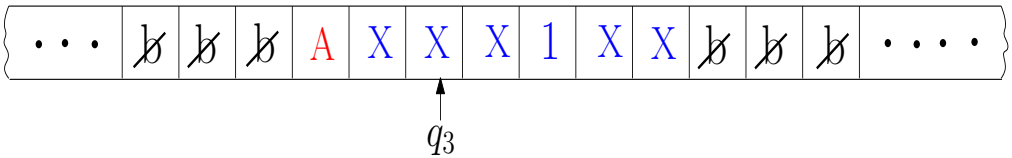
$$Move: \delta(q_0, 1) = (q_1, X, R)$$



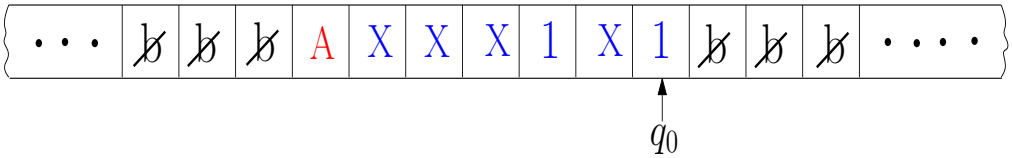
$$Move: \delta(q_1, \not b) = (q_3, \not b, \mathrm{L})$$



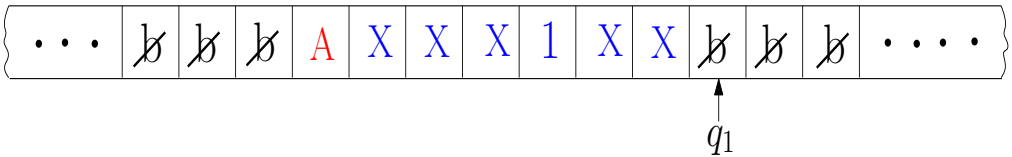
$$Move: \delta(q_3, Z) = (q_3, Z, L) \text{ where } Z \in \{\mathrm{X}, 1, \mathrm{A}, \mathrm{B}\}$$



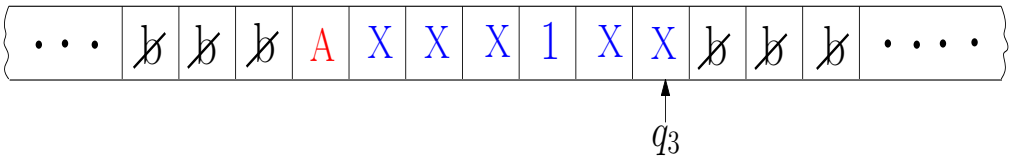
Turing Machines (A input output device)



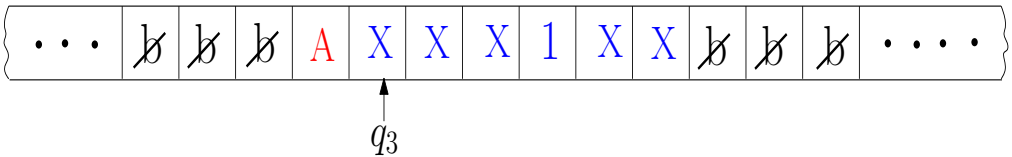
$$Move: \delta(q_0, 1) = (q_1, X, R)$$



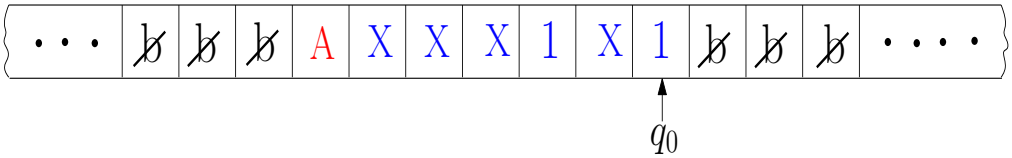
$$Move: \delta(q_1, \not\vdash) = (q_3, \not\vdash, \mathrm{L})$$



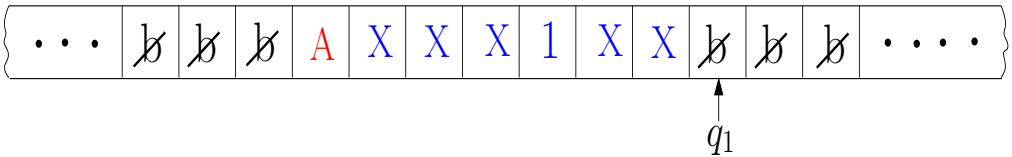
$$Move: \delta(q_3, Z) = (q_3, Z, L) \text{ where } Z \in \{\mathrm{X}, 1, \mathrm{A}, \mathrm{B}\}$$



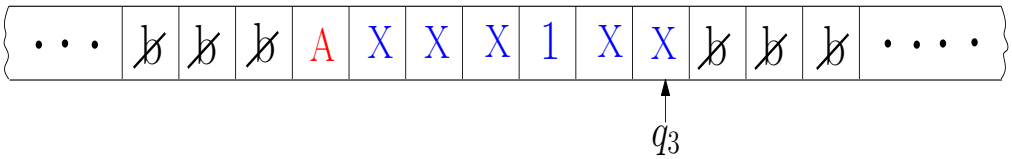
Turing Machines (A input output device)



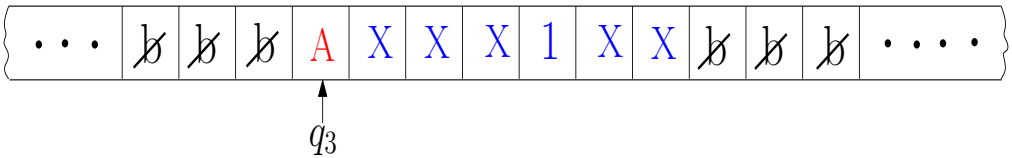
$$Move: \delta(q_0, 1) = (q_1, X, R)$$



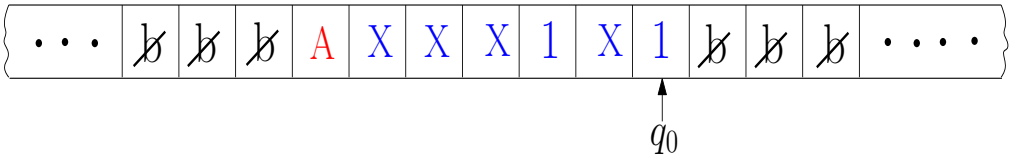
$$Move: \delta(q_1, \not\vdash) = (q_3, \not\vdash, \mathrm{L})$$



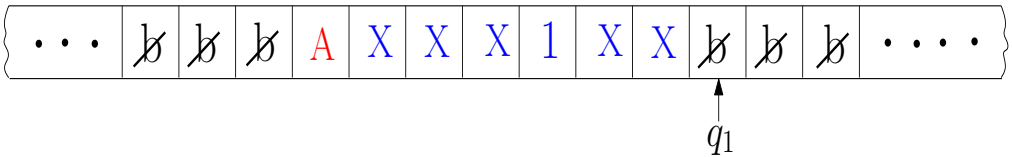
$$Move: \delta(q_3, Z) = (q_3, Z, L) \text{ where } Z \in \{\mathrm{X}, 1, \mathrm{A}, \mathrm{B}\}$$



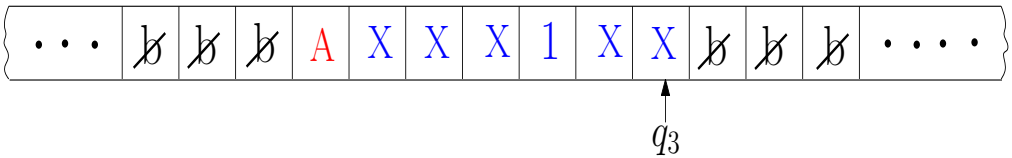
Turing Machines (A input output device)



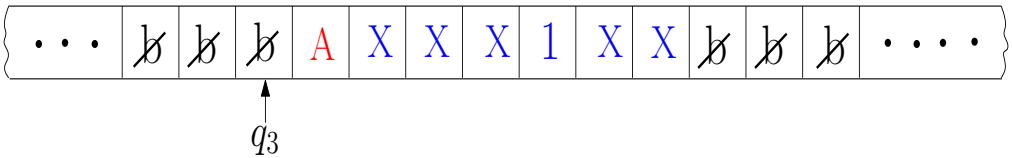
$$Move: \delta(q_0, 1) = (q_1, X, R)$$



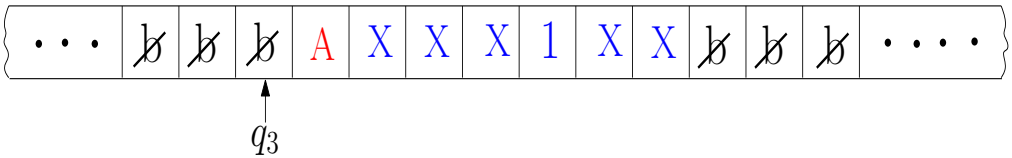
$$Move: \delta(q_1, \not\vdash) = (q_3, \not\vdash, \mathrm{L})$$



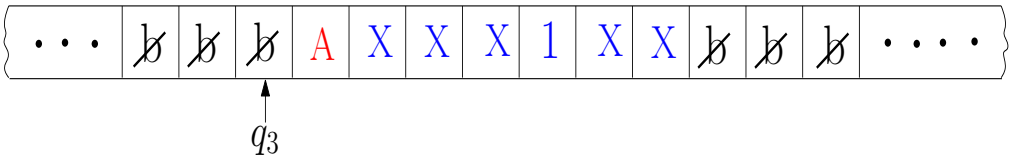
$$Move: \delta(q_3, Z) = (q_3, Z, L) \text{ where } Z \in \{\mathrm{X}, 1, \mathrm{A}, \mathrm{B}\}$$



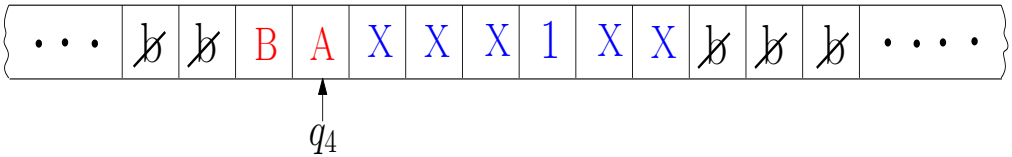
Turing Machines (A input output device)



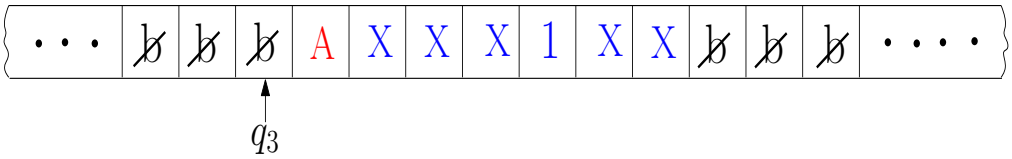
Turing Machines (A input output device)



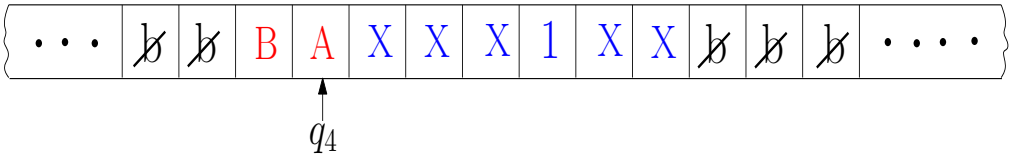
$$Move: \delta(q_3, \not b) = (q_4, B, R)$$



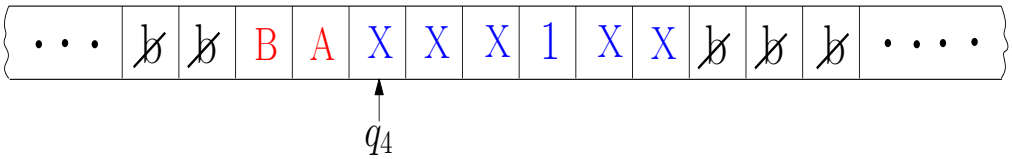
Turing Machines (A input output device)



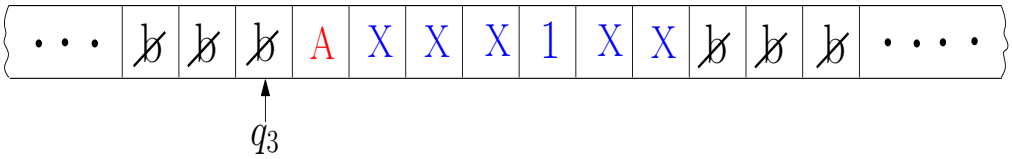
$$Move: \delta(q_3, \not b) = (q_4, B, R)$$



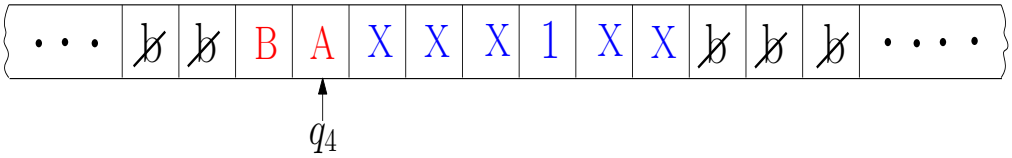
$$Move: \delta(q_4, Y) = (q_4, Y, R) \text{ where } Y \in \{\text{X, A, B}\}$$



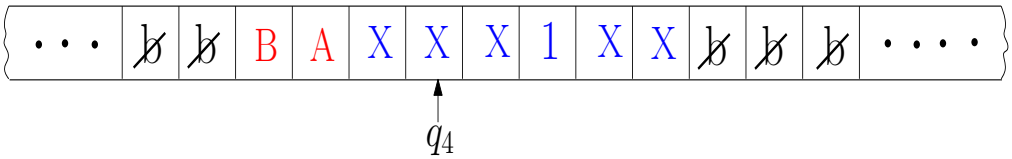
Turing Machines (A input output device)



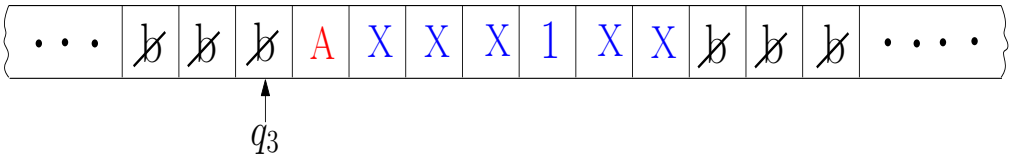
$$Move: \delta(q_3, \not b) = (q_4, B, R)$$



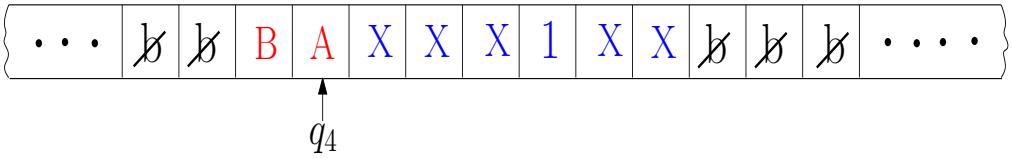
$$Move: \delta(q_4, Y) = (q_4, Y, R) \text{ where } Y \in \{\text{X, A, B}\}$$



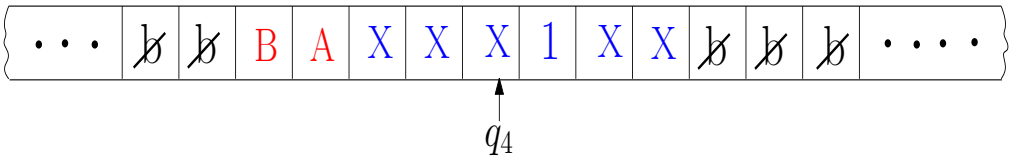
Turing Machines (A input output device)



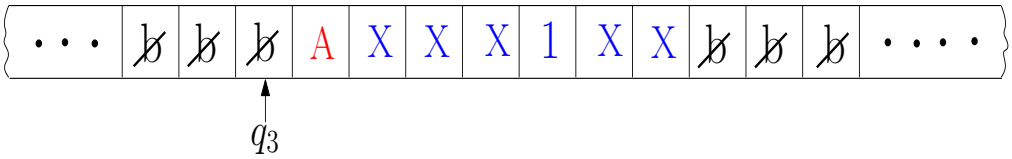
$$Move: \delta(q_3, \not b) = (q_4, B, R)$$



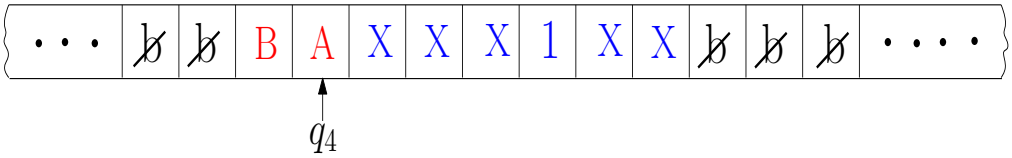
$$Move: \delta(q_4, Y) = (q_4, Y, R) \text{ where } Y \in \{\text{X, A, B}\}$$



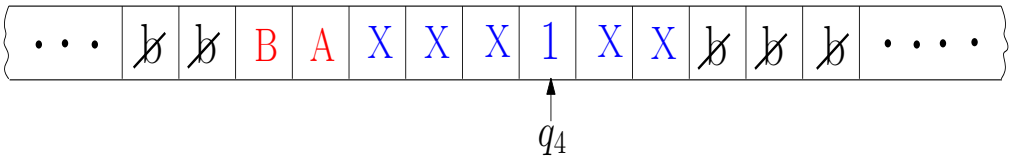
Turing Machines (A input output device)



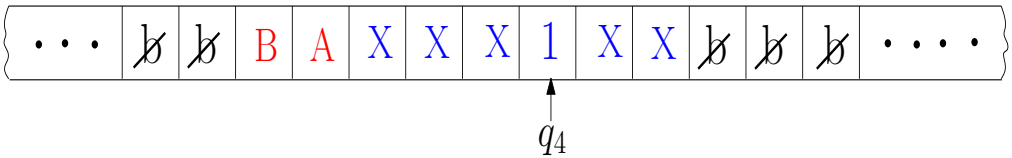
$$Move: \delta(q_3, \not b) = (q_4, B, R)$$



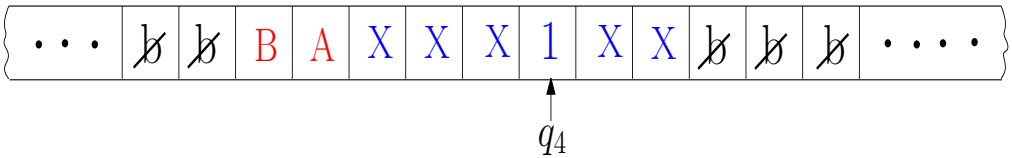
$$Move: \delta(q_4, Y) = (q_4, Y, R) \text{ where } Y \in \{\text{X, A, B}\}$$



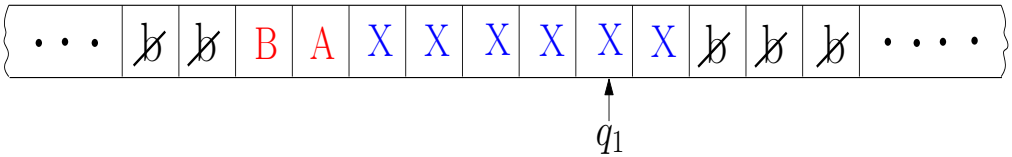
Turing Machines (A input output device)



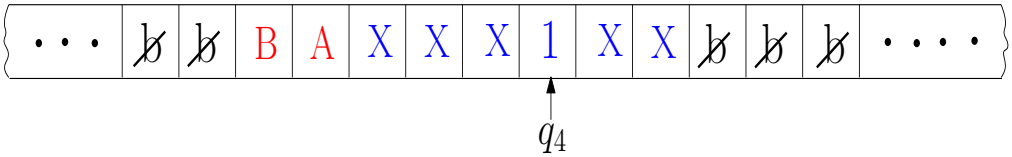
Turing Machines (A input output device)



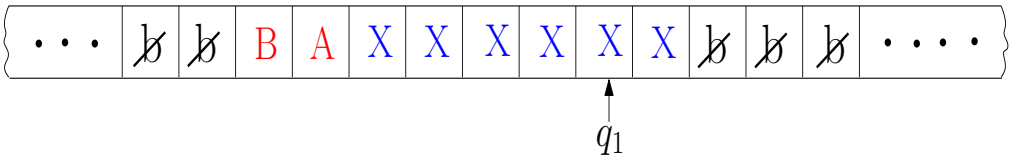
$$Move: \delta(q_4, 1) = (q_1, X, R)$$



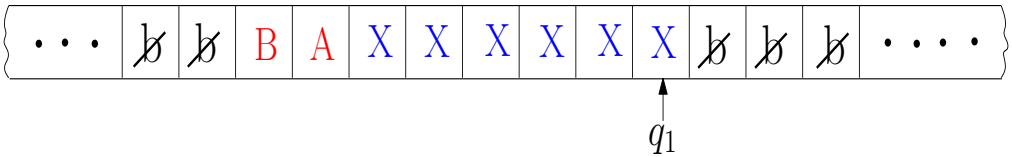
Turing Machines (A input output device)



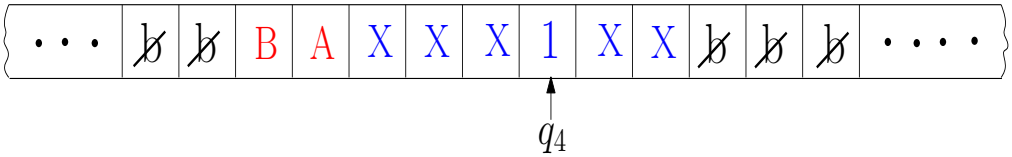
$$Move: \delta(q_4, 1) = (q_1, X, R)$$



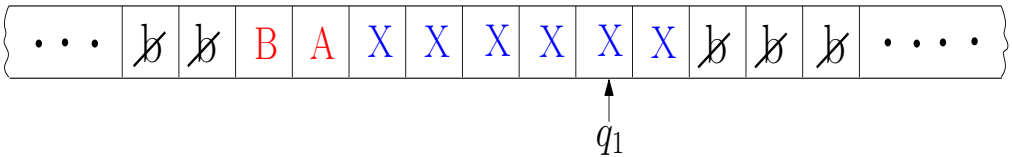
$$Move: \delta(q_1, X) = (q_1, X, R)$$



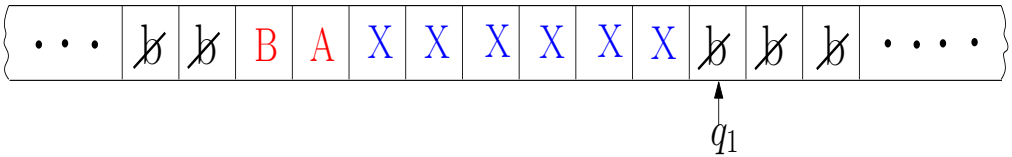
Turing Machines (A input output device)



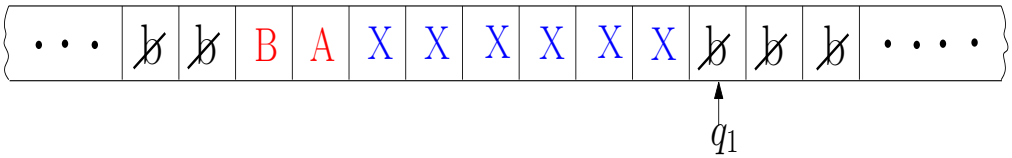
$$Move: \delta(q_4, 1) = (q_1, X, R)$$



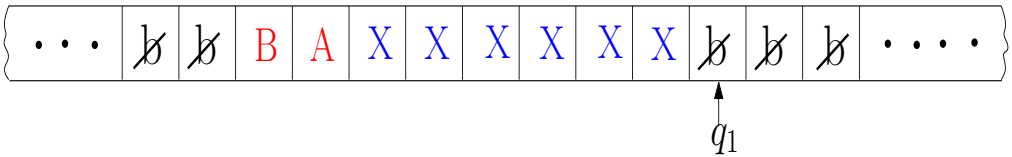
$$Move: \delta(q_1, X) = (q_1, X, R)$$



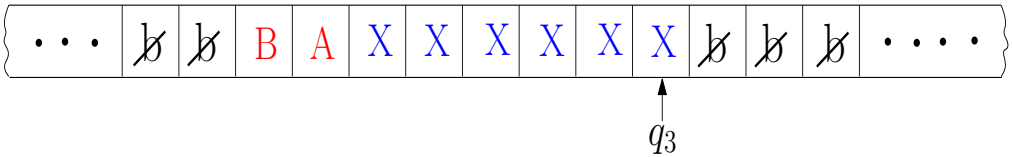
Turing Machines (A input output device)



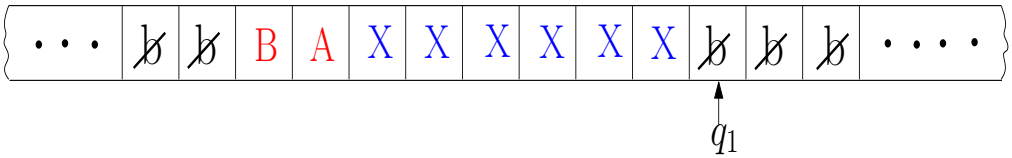
Turing Machines (A input output device)



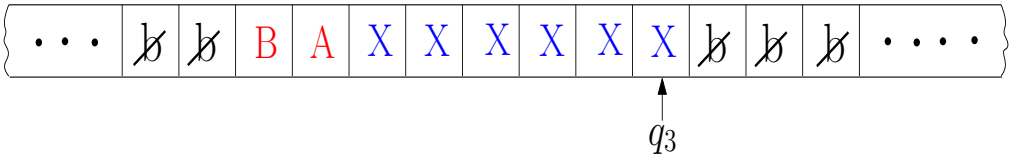
$$Move : \delta(q_1, \not b) = (q_3, \not b, \mathrm{L})$$



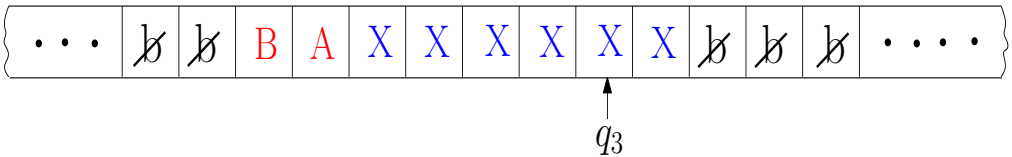
Turing Machines (A input output device)



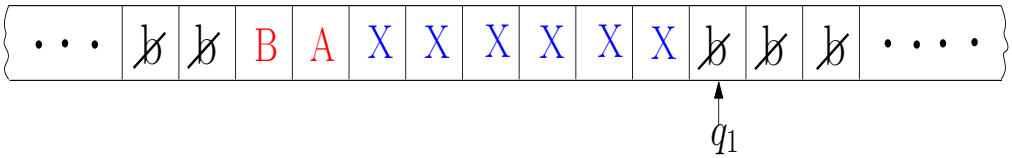
$$Move: \delta(q_1, \not b) = (q_3, \not b, \mathrm{L})$$



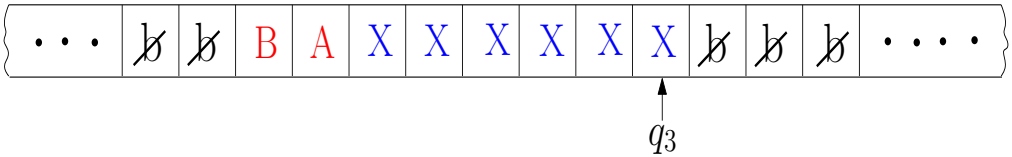
$$Move: \delta(q_3, Z) = (q_3, Z, L) \text{ where } Z \in \{\mathrm{X}, 1, \mathrm{A}, \mathrm{B}\}$$



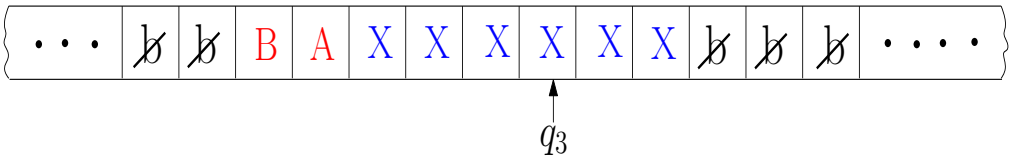
Turing Machines (A input output device)



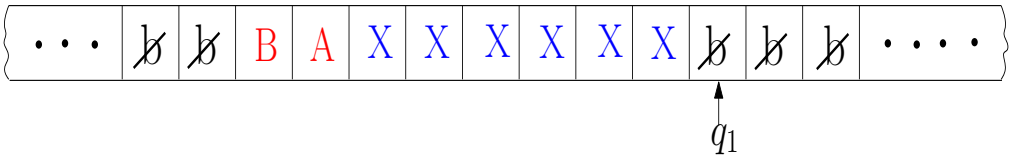
$$Move: \delta(q_1, \not b) = (q_3, \not b, \mathrm{L})$$



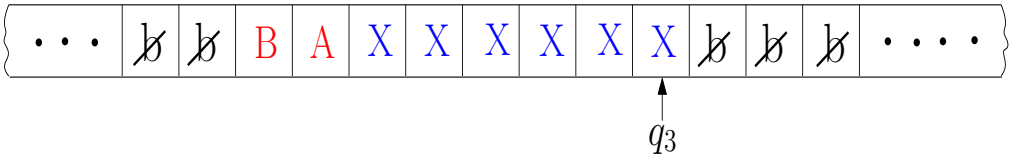
$$Move: \delta(q_3, Z) = (q_3, Z, L) \text{ where } Z \in \{\mathrm{X}, 1, \mathrm{A}, \mathrm{B}\}$$



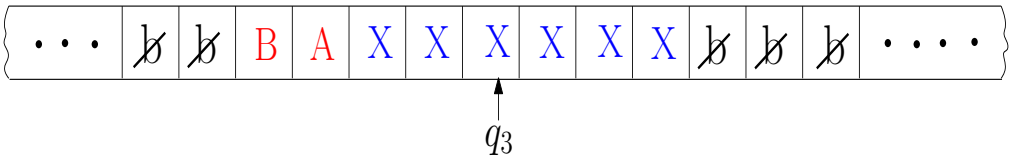
Turing Machines (A input output device)



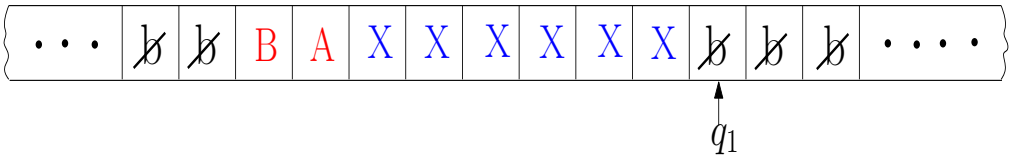
$$Move: \delta(q_1, \not b) = (q_3, \not b, \mathrm{L})$$



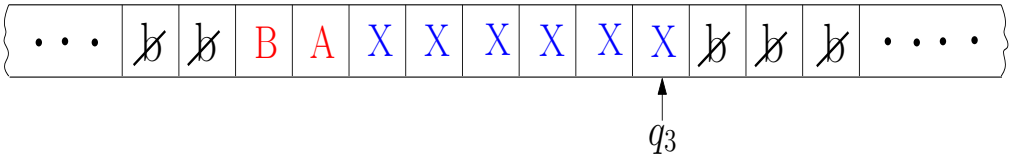
$$Move: \delta(q_3, Z) = (q_3, Z, L) \text{ where } Z \in \{\mathrm{X}, 1, \mathrm{A}, \mathrm{B}\}$$



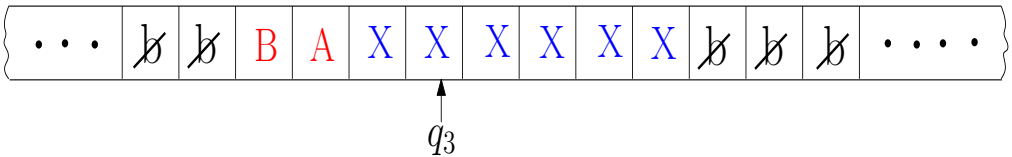
Turing Machines (A input output device)



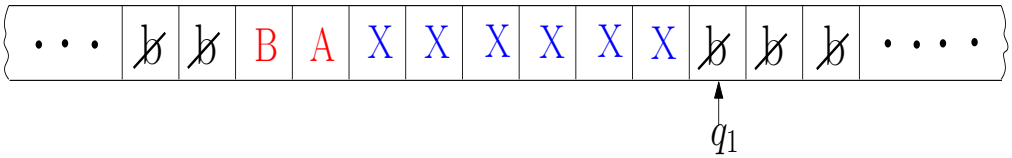
$$Move: \delta(q_1, \not b) = (q_3, \not b, \mathrm{L})$$



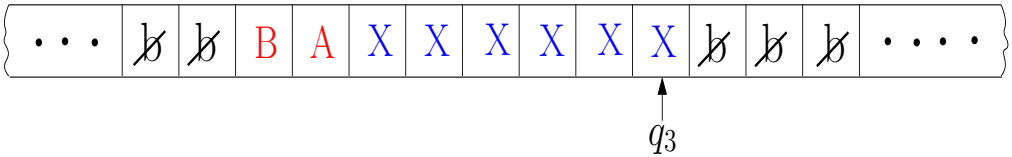
$$Move: \delta(q_3, Z) = (q_3, Z, L) \text{ where } Z \in \{\mathrm{X}, 1, \mathrm{A}, \mathrm{B}\}$$



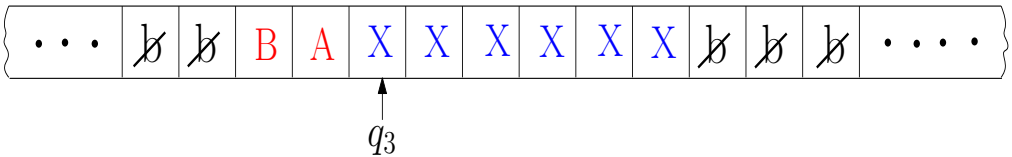
Turing Machines (A input output device)



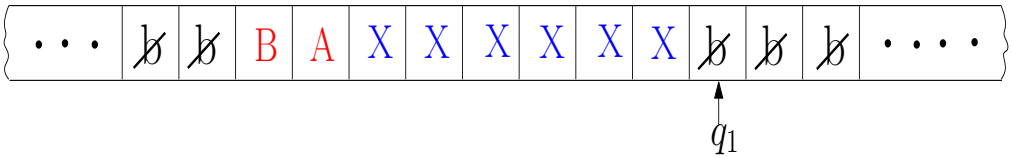
$$Move: \delta(q_1, \not b) = (q_3, \not b, \mathrm{L})$$



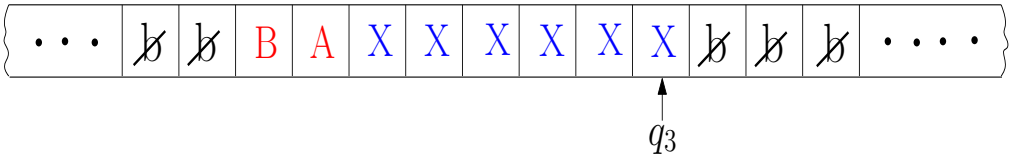
$$Move: \delta(q_3, Z) = (q_3, Z, L) \text{ where } Z \in \{\mathrm{X}, 1, \mathrm{A}, \mathrm{B}\}$$



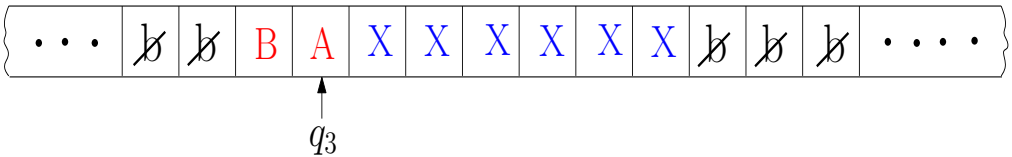
Turing Machines (A input output device)



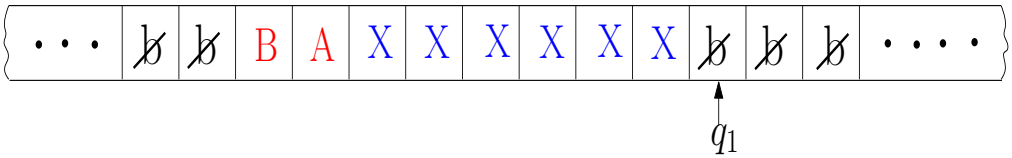
$$Move: \delta(q_1, \not b) = (q_3, \not b, \mathrm{L})$$



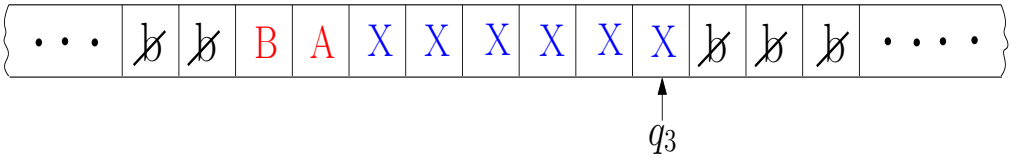
$$Move: \delta(q_3, Z) = (q_3, Z, L) \text{ where } Z \in \{\mathrm{X}, 1, \mathrm{A}, \mathrm{B}\}$$



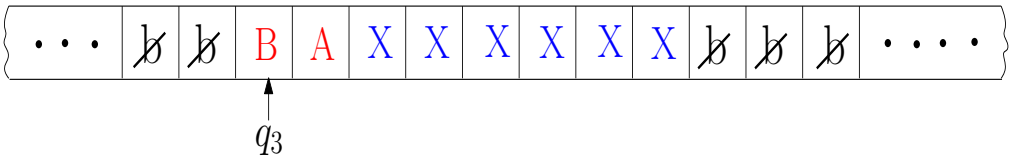
Turing Machines (A input output device)



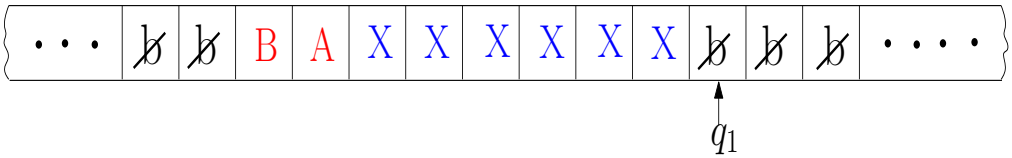
$$Move: \delta(q_1, \not b) = (q_3, \not b, \mathrm{L})$$



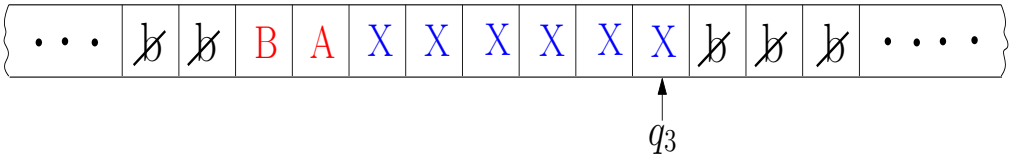
$$Move: \delta(q_3, Z) = (q_3, Z, L) \text{ where } Z \in \{\mathrm{X}, 1, \mathrm{A}, \mathrm{B}\}$$



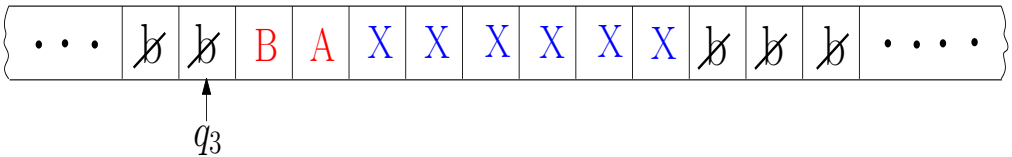
Turing Machines (A input output device)



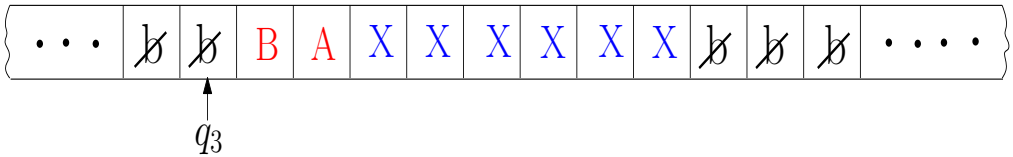
$$Move: \delta(q_1, \not b) = (q_3, \not b, \mathrm{L})$$



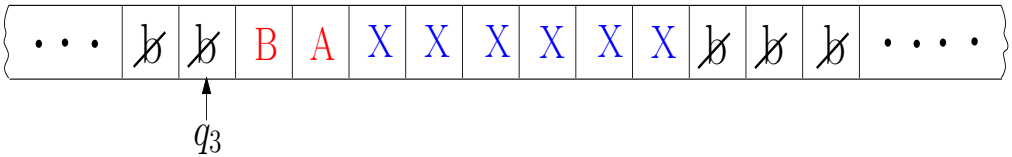
$$Move: \delta(q_3, Z) = (q_3, Z, L) \text{ where } Z \in \{\mathrm{X}, 1, \mathrm{A}, \mathrm{B}\}$$



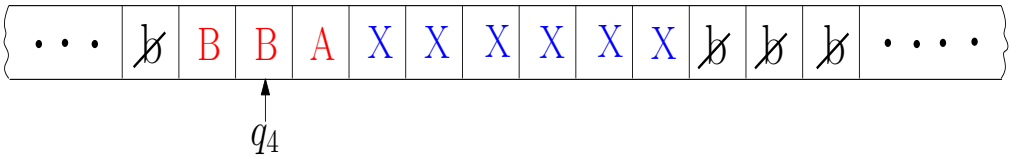
Turing Machines (A input output device)



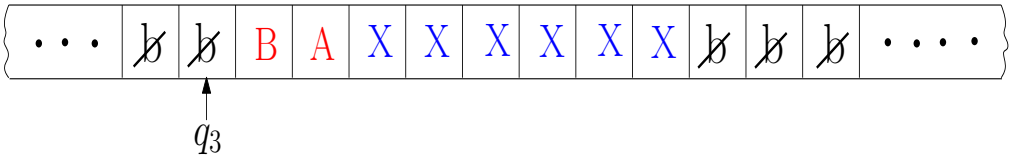
Turing Machines (A input output device)



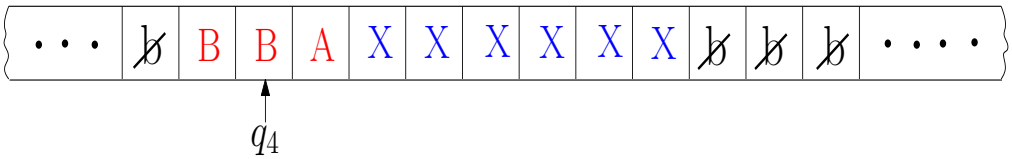
$$Move: \delta(q_3, \not b) = (q_4, B, R)$$



Turing Machines (A input output device)

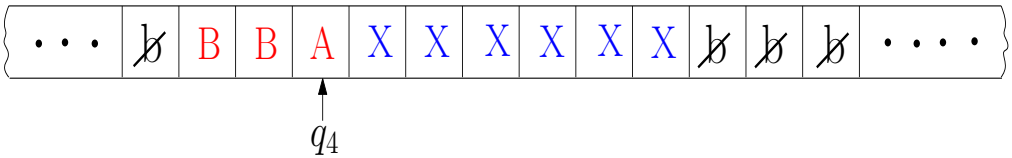


$$Move: \delta(q_3, \not b) = (q_4, B, R)$$

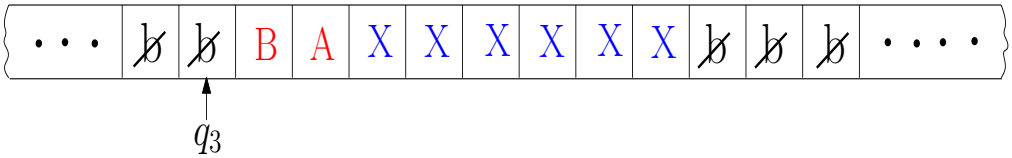


$$Move: \delta(q_4, Y) = (q_4, Y, R) \text{ where } Y \in \{\text{X, A, B}\}$$

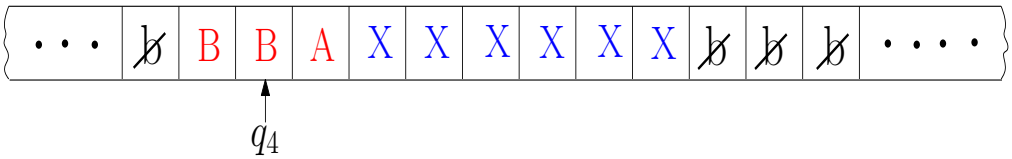
$$Move: \delta(q_4, \not b) = (\text{H}, \not b, \text{-})$$



Turing Machines (A input output device)

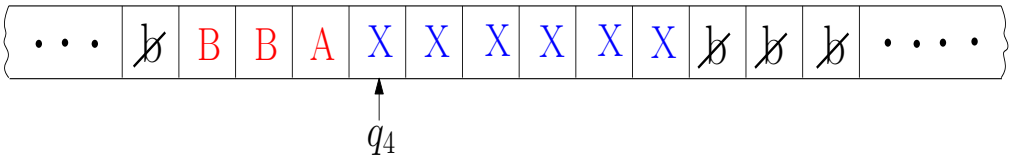


$$Move: \delta(q_3, \not b) = (q_4, B, R)$$

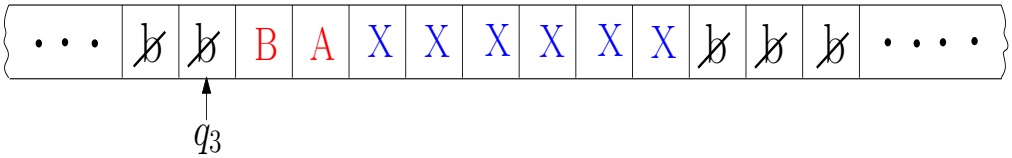


$$Move: \delta(q_4, Y) = (q_4, Y, R) \text{ where } Y \in \{\text{X, A, B}\}$$

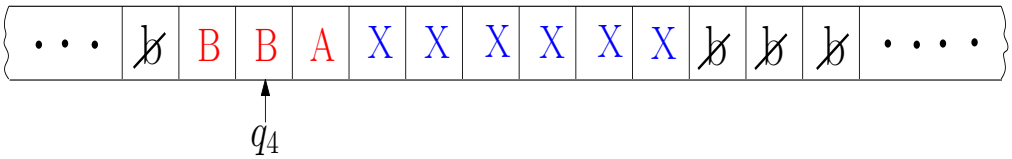
$$Move: \delta(q_4, \not b) = (\text{H}, \not b, \text{-})$$



Turing Machines (A input output device)

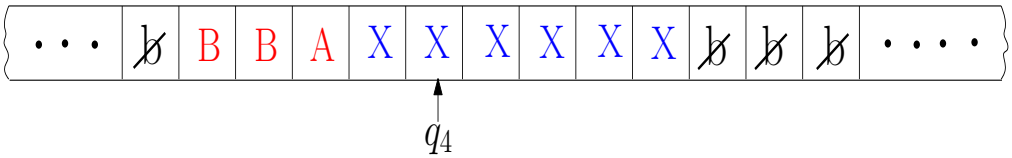


$$Move: \delta(q_3, \not b) = (q_4, B, R)$$

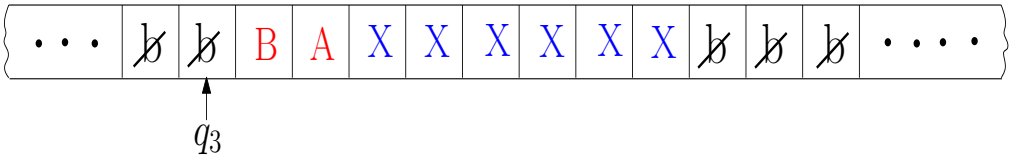


$$Move: \delta(q_4, Y) = (q_4, Y, R) \text{ where } Y \in \{\text{X, A, B}\}$$

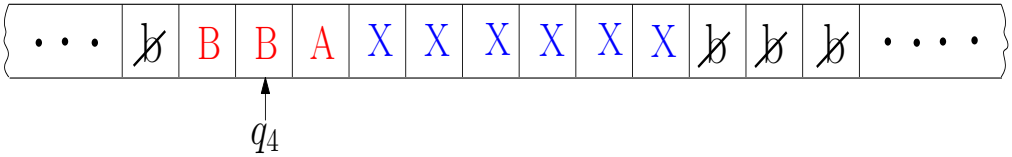
$$Move: \delta(q_4, \not b) = (\text{H}, \not b, \text{-})$$



Turing Machines (A input output device)

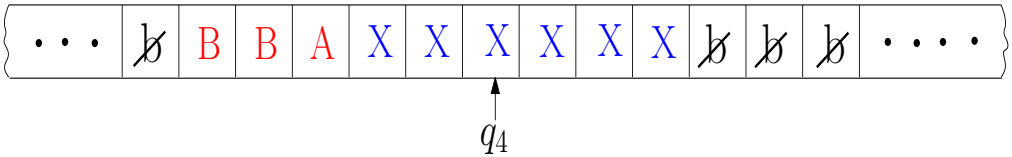


$$Move: \delta(q_3, \not b) = (q_4, B, R)$$

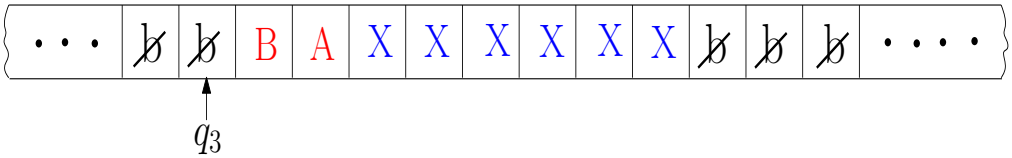


$$Move: \delta(q_4, Y) = (q_4, Y, R) \text{ where } Y \in \{\text{X, A, B}\}$$

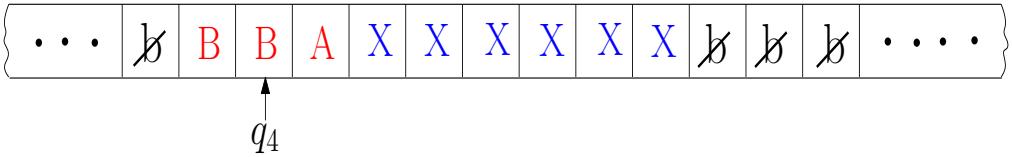
$$Move: \delta(q_4, \not b) = (\text{H}, \not b, \text{-})$$



Turing Machines (A input output device)

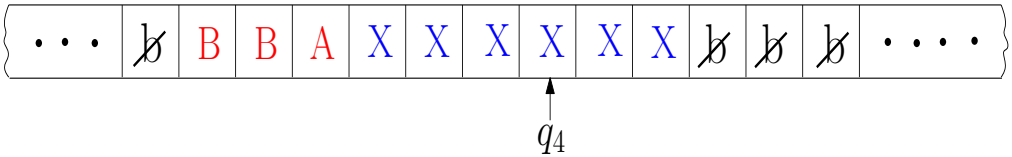


$$Move: \delta(q_3, \not b) = (q_4, B, R)$$

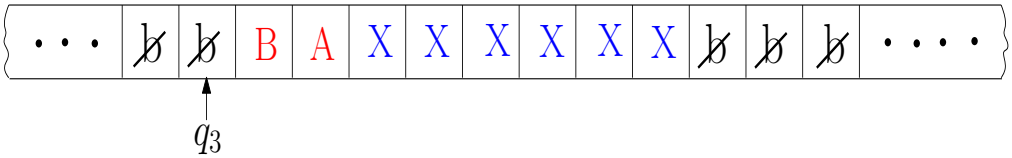


$$Move: \delta(q_4, Y) = (q_4, Y, R) \text{ where } Y \in \{\text{X, A, B}\}$$

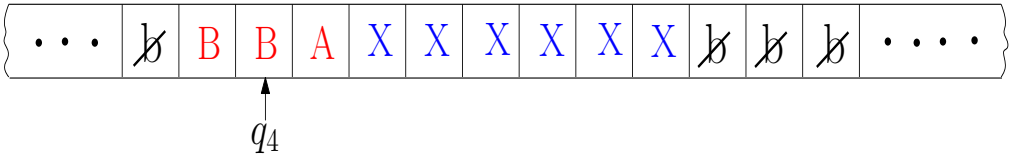
$$Move: \delta(q_4, \not b) = (\text{H}, \not b, \text{-})$$



Turing Machines (A input output device)

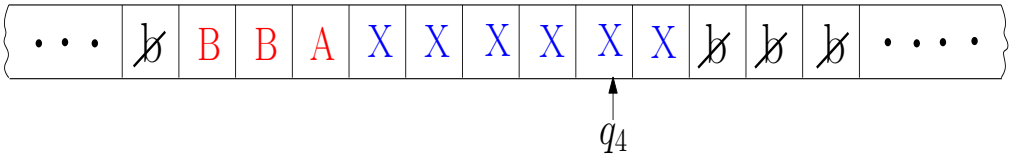


$$Move: \delta(q_3, \not b) = (q_4, B, R)$$

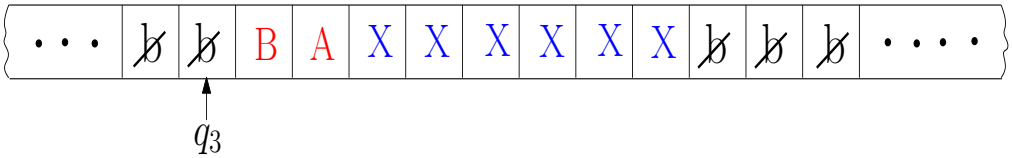


$$Move: \delta(q_4, Y) = (q_4, Y, R) \text{ where } Y \in \{\text{X, A, B}\}$$

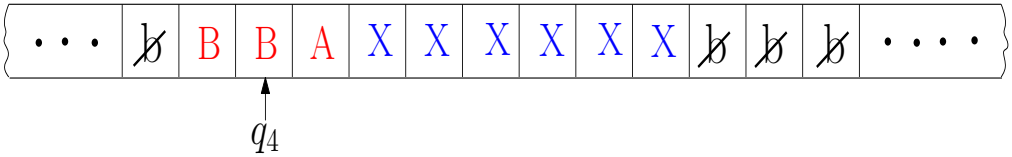
$$Move: \delta(q_4, \not b) = (\text{H}, \not b, \text{-})$$



Turing Machines (A input output device)

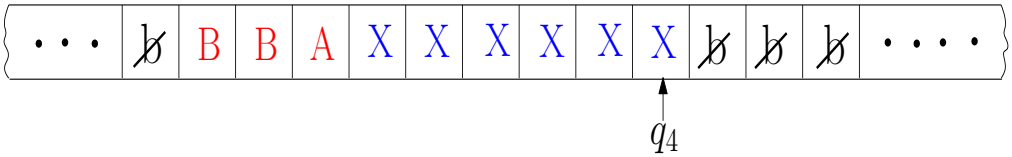


$$Move: \delta(q_3, \not b) = (q_4, B, R)$$

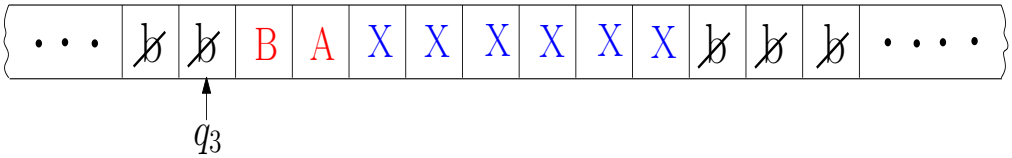


$$Move: \delta(q_4, Y) = (q_4, Y, R) \text{ where } Y \in \{\text{X, A, B}\}$$

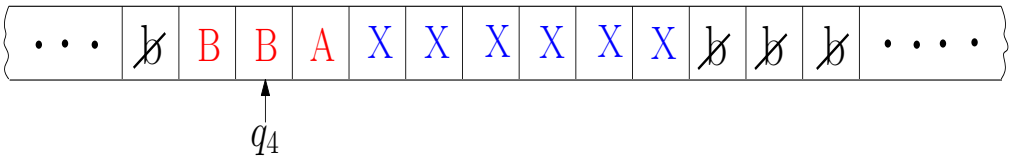
$$Move: \delta(q_4, \not b) = (\text{H}, \not b, \text{-})$$



Turing Machines (A input output device)

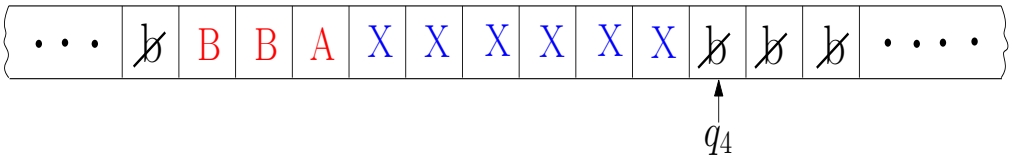


$$Move: \delta(q_3, \not b) = (q_4, B, R)$$

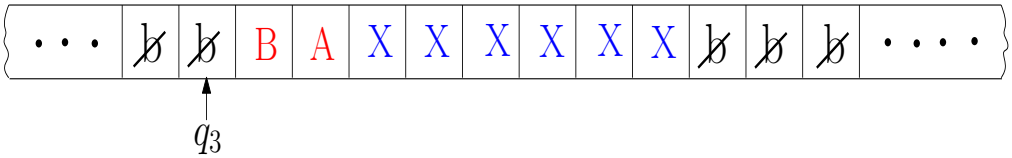


$$Move: \delta(q_4, Y) = (q_4, Y, R) \text{ where } Y \in \{\text{X, A, B}\}$$

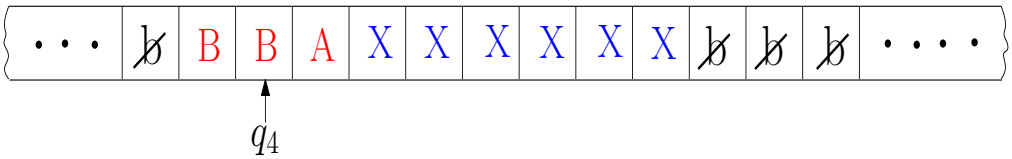
$$Move: \delta(q_4, \not b) = (\text{H}, \not b, \text{-})$$



Turing Machines (A input output device)

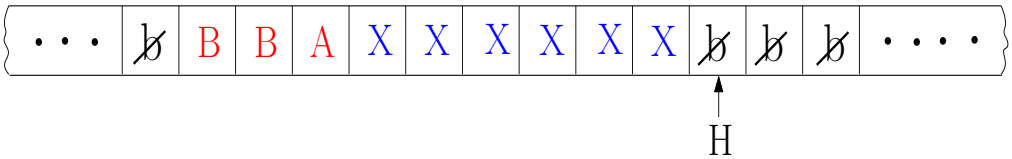


$$Move: \delta(q_3, \not{b}) = (q_4, B, R)$$



$$Move: \delta(q_4, Y) = (q_4, Y, R) \text{ where } Y \in \{\text{X, A, B}\}$$

$$Move: \delta(q_4, \not{b}) = (\text{H}, \not{b}, \text{-})$$



Turing Machines (A input output device)

- So, the Turing machine M = (Q, Σ, Γ, δ, q_0, ∦, F) where
  - States $Q = \{q_0, q_1, q_2, q_3, q_4\}$
  - Input alphabet $\Sigma = \{1\}$
  - Tape alphabet $\Gamma = \{1, \text{X}, \text{A}, \text{B}, \not\#\}$
  - Mapping δ : $\delta(q_0, 1) = (q_1, X, R)$ $\quad$ $\delta(q_1, 1) = (q_0, 1, R)$

    $\delta(q_1, \not\#) = (q_3, \not\#, \text{L})$ $\quad$ $\delta(q_3, \not\#) = (q_4, B, R)$

    $\delta(q_4, 1) = (q_1, X, R)$ $\quad$ $\delta(q_1, X) = (q_1, X, R)$

    $\delta(q_0, X) = (q_0, X, R)$ $\quad$ $\delta(q_0, \not\#) = (q_2, \not\#, \text{L})$

    $\delta(q_2, \not\#) = (q_4, A, R)$

    $\delta(q_3, Z) = (q_3, Z, L)$ where $Z \in \{X, 1, A, B\}$

    $\delta(q_4, Y) = (q_4, Y, R)$ where $Y \in \{X, A, B\}$

    $\delta(q_2, Z) = (q_2, Z, L)$ where $Z \in \{X, 1, A, B\}$

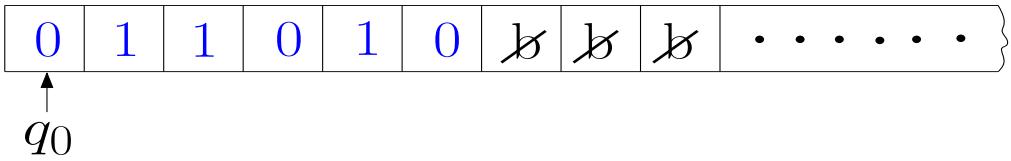
    $\delta(q_4, \not\#) = (\text{H}, \not\#, \text{-})$

Turing Machines (A accepting device)

- Turing machines can be viewed as an accepters.
- When we look at it as an accepter, we fixed the left end, right end it is infinite.

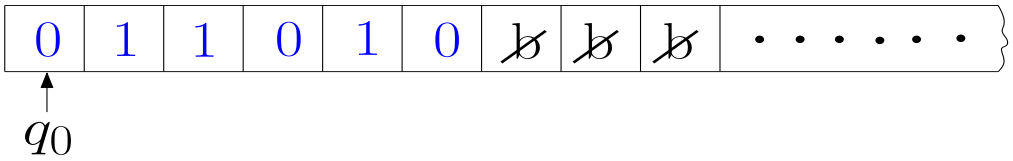
Turing Machines (A accepting device)

- Turing machines can be viewed as an accepters.
- When we look at it as an accepter, we fixed the left end, right end it is infinite.



Turing Machines (A accepting device)

- Turing machines can be viewed as an accepters.
- When we look at it as an accepter, we fixed the left end, right end it is infinite.



- When the machine starts, initial state reading the leftmost symbol.
- When it reaches a final state, without loss of generality we assume that it halts.
- So, when reaches a final state it will stop and accept the input.

Instantaneous Descriptions for Turing Machines

- In order to describe formally what a Turing machine does, we need to develop a notation for configurations or ***instantaneous descriptions*** (ID's).
- Since a TM has an infinitely long tape, we might imagine that it is impossible to describe the configurations of a TM.
- However, after any finite number of moves, the TM can have visited only a finite number of cells, even though the number of cells visited can eventually grow beyond any finite limit.
- Thus, in every ID, there is an infinite prefix and infinite suffix of cells that have never been visited.
- These cells must all hold either blanks or one of the finite number of input symbols.

Instantaneous Descriptions for Turing Machines

- We thus show in an ID only the cells between the leftmost and the rightmost non blanks.
- Thus, we shall use the string $X_1 X_2 \cdots X_{i-1} q X_i X_{i+1} \cdots X_n$ to represents an ID in which
  - q is the state of the Turing machine
  - The tape head is scanning the i^{th} symbol from the left
  - $X_1 X_2 \cdots X_n$ is the portion of the tape between the leftmost and the rightmost nonblank.

Instantaneous Descriptions for Turing Machines

- We describe moves of a Turing machine $\mathrm{M} = (\mathrm{Q}, \Sigma, \Gamma, \delta, q_0, \not b, \mathrm{F})$ by the notation "$\vdash$"
- Suppose $\delta(q, X_i) = (p, Y, L)$; i.e., the next move is leftward. Then,

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

- Suppose $\delta(q, X_i) = (p, Y, R)$; i.e., the next move is rightward. Then,

$$X_1 X_2 \cdots X_{i-1} q X_i X_{i+1} \cdots X_n \vdash X_1 X_2 \cdots X_{i-1} Y p X_{i+1} \cdots X_n$$

Instantaneous Descriptions for Turing Machines

- We describe moves of a Turing machine M = (Q, Σ, Γ, δ, q_0, ∅, F) by the notation "⊢"
- Suppose $\delta(q, X_i) = (p, Y, L)$; i.e., the next move is leftward. Then,

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

- Suppose $\delta(q, X_i) = (p, Y, R)$; i.e., the next move is rightward. Then,

$$X_1 X_2 \cdots X_{i-1} q X_i X_{i+1} \cdots X_n \vdash X_1 X_2 \cdots X_{i-1} Y p X_{i+1} \cdots X_n$$

- Consider the initial state of a TM is q_0 and the input is w, then initial ID = $q_0 w$
  Now we assume that once it reaches a final state it halts. So, it halts with some ID $\alpha_1 q_f \alpha_2$ where $q_f \in F$ and $\alpha_1, \alpha_2 \in \Gamma^*$

Turing Machines as Language Accepters

- Consider a string w is written on the tape, with blanks filling out the unused portions.
- The machine is stated in the initial state q_0 with the read-write head positioned on the leftmost symbol of w.
- If, after a sequence of moves, the Turing machine enters a final state and halts, then w is considered to be accepted.

Turing Machines as Language Accepters

- Consider a string w is written on the tape, with blanks filling out the unused portions.
- The machine is stated in the initial state q_0 with the read-write head positioned on the leftmost symbol of w.
- If, after a sequence of moves, the Turing machine enters a final state and halts, then w is considered to be accepted.

- **Definition :** Let M = (Q, Σ, Γ, δ, q_0, $\not\emptyset$, F) be a Turing machine. Then the language accepted by M is,

$$L(M) = \{w | w \in \Sigma^* : q_0 w \vdash^* \alpha_1 q_f \alpha_2 \text{ where } q_f \in F, \alpha_1, \alpha_2 \in \Gamma^*\}$$

Turing Machines as Language Accepters

- Consider a string w is written on the tape, with blanks filling out the unused portions.
- The machine is stated in the initial state q_0 with the read-write head positioned on the leftmost symbol of w.
- If, after a sequence of moves, the Turing machine enters a final state and halts, then w is considered to be accepted.

- **Definition :** Let M = (Q, Σ, Γ, δ, q_0, $\not\emptyset$, F) be a Turing machine. Then the language accepted by M is,

$$L(M) = \{w | w \in \Sigma^* : q_0 w \vdash^* \alpha_1 q_f \alpha_2 \text{ where } q_f \in F, \alpha_1, \alpha_2 \in \Gamma^*\}$$

- The set of languages we can accept using a Turing machine is often called the *recursively enumerable languages* or RE languages.

Turing Machines as Language Accepters

Example 1: For $\Sigma = \{a, b\}$, design a Turing machine that accepts

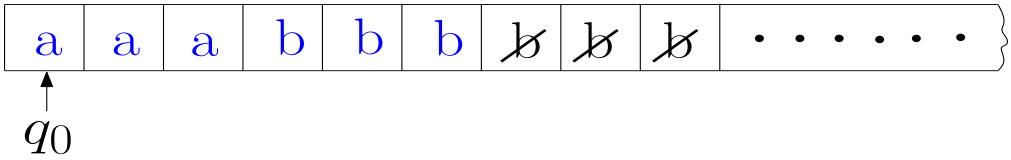
$$L = \{a^n b^n : n \geq 1\}$$

Turing Machines as Language Accepters

Example 1: For $\Sigma = \{a, b\}$, design a Turing machine that accepts

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

Solution : We solve the problem in the following fashion :

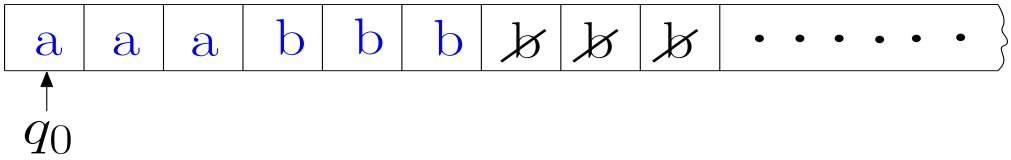


Turing Machines as Language Accepters

Example 1: For $\Sigma = \{a, b\}$, design a Turing machine that accepts

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

Solution : We solve the problem in the following fashion :



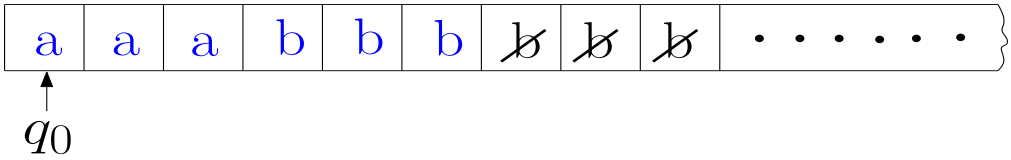
- Staring at the leftmost a, we check it off by replacing it with some symbol, say X.

Turing Machines as Language Accepters

Example 1: For $\Sigma = \{a, b\}$, design a Turing machine that accepts

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

Solution : We solve the problem in the following fashion :



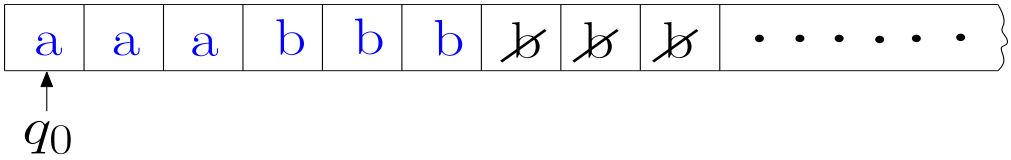
- Staring at the leftmost a, we check it off by replacing it with some symbol, say X.
- Then let the read-write head travel right to find the leftmost b, which in turn is checked off by replacing it with another symbol, say Y.

Turing Machines as Language Accepters

Example 1: For $\Sigma = \{a, b\}$, design a Turing machine that accepts

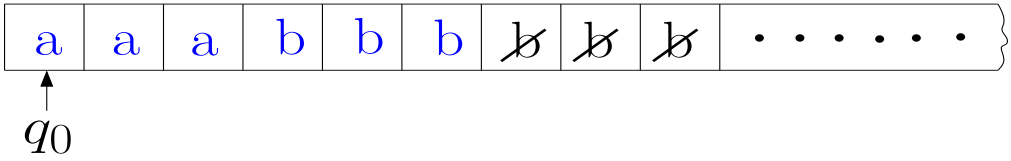
$$L = \{a^n b^n : n \geq 1\}$$

Solution : We solve the problem in the following fashion :



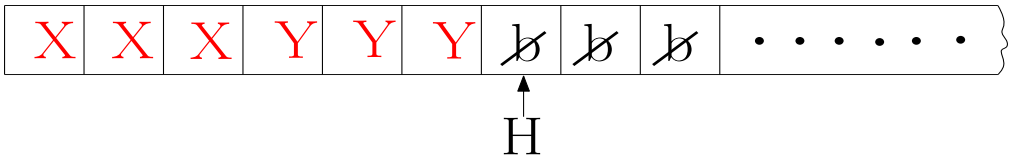
- Staring at the leftmost a, we check it off by replacing it with some symbol, say X.
- Then let the read-write head travel right to find the leftmost b, which in turn is checked off by replacing it with another symbol, say Y.
- After that, we go left again to the leftmost a, replace it with an X, then move to the leftmost b and replace it with Y, and so on.
- Traveling back and forth this way, we match each a with a corresponding b. If after some time no a's or b's remain, then the string must be in L.

Turing Machines as Language Accepters



- So, the Turing machine M = (Q, Σ, Γ, δ, q_0, ⊬, F) where
  - $Q = \{q_0, q_1, q_2, q_3, q_4\}$,
  - $\Sigma = \{a, b\}$,
  - $\Gamma = \{a, b, X, Y,, \not\vdash\}$
  - $F = \{q_4\}$
  - δ :

$$\begin{aligned}
&\delta(q_0, a) = (q_1, X, R), && \delta(q_1, a) = (q_1, a, R),\\
&\delta(q_1, Y) = (q_1, Y, R), && \delta(q_1, b) = (q_2, Y, L),\\
&\delta(q_2, Y) = (q_2, Y, L), && \delta(q_2, a) = (q_2, a, L),\\
&\delta(q_2, X) = (q_0, X, R), && \delta(q_0, y) = (q_3, Y, R),\\
&\delta(q_3, Y) = (q_3, Y, R), && \delta(q_3, \not\vdash) = (q_4, \text{H}, \text{-})
\end{aligned}$$



Turing Machines (Part-II)

Raju Hazari

Department of Computer Science and Engineering
National Institute of Technology Calicut

September 10, 2021

Transition Diagrams for Turing Machines

- We can represent the transitions of a Turing machine pictorially.
- A transition diagram consists of a set of nodes corresponding to the states of the TM.
- An *arc* from state q to state p is labeled by one or more items of the form X/Y D, where X and Y are tape symbols, and D is a direction, either L or R.
  - That is, whenever $\delta(q, X) = (p, Y, D)$, we find the label X/Y D on the arc from q to p.
- The direction D can be pictorially represent by $\leftarrow$ for "left" and $\rightarrow$ for "right".

Transition Diagrams for Turing Machines

- **Example :** Consider a TM which accept the language $L = \{0^n 1^n | n \geq 1\}$. Now the formal specification of the TM M is
  $$M = (\{q_0, q_1, q_2, q_3, q_4\}, \{0, 1\}, \{0, 1, X, Y, B\}, \delta, q_0, B, \{q_4\})$$
  where δ is given by

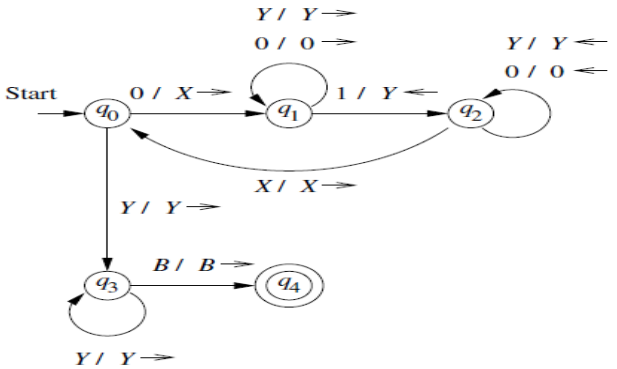
State	Symbol 0	1	X	Y	B
q_0	(q_1, X, R)	–	–	(q_3, Y, R)	–
q_1	$(q_1, 0, R)$	(q_2, Y, L)	–	(q_1, Y, R)	–
q_2	$(q_2, 0, L)$	–	(q_0, X, R)	(q_2, Y, L)	–
q_3	–	–	–	(q_3, Y, R)	(q_4, B, R)
q_4	–	–	–	–	–

Transition Diagrams for Turing Machines

State	Symbol 0	1	X	Y	B
q_0	(q_1, X, R)	–	–	(q_3, Y, R)	–
q_1	$(q_1, 0, R)$	(q_2, Y, L)	–	(q_1, Y, R)	–
q_2	$(q_2, 0, L)$	–	(q_0, X, R)	(q_2, Y, L)	–
q_3	–	–	–	(q_3, Y, R)	(q_4, B, R)
q_4	–	–	–	–	–

Transition Diagrams for Turing Machines

State	Symbol 0	1	X	Y	B
q_0	(q_1, X, R)	–	–	(q_3, Y, R)	–
q_1	$(q_1, 0, R)$	(q_2, Y, L)	–	(q_1, Y, R)	–
q_2	$(q_2, 0, L)$	–	(q_0, X, R)	(q_2, Y, L)	–
q_3	–	–	–	(q_3, Y, R)	(q_4, B, R)
q_4	–	–	–	–	–



Transition diagram for a TM that accepts strings of the form 0^n1^n

Techniques for TM construction

1. Storage in the State :

Techniques for TM construction

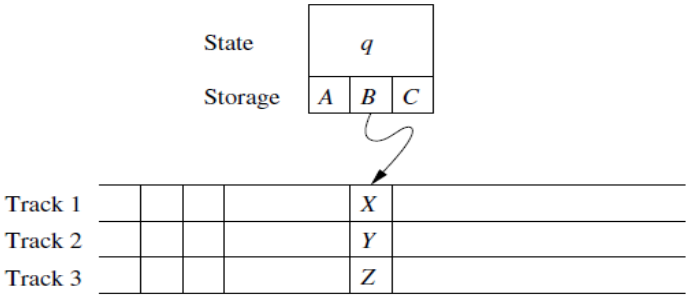
1. Storage in the State :

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

Techniques for TM construction

1. Storage in the State :

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



- Here, the finite control consisting of not only a "control" state q, but three data elements A, B and C.
- So, we may think of the state as a tuple.
- This way allows us to describe transitions in a more systematic way.

Techniques for TM construction

- **Example :** We shall use the technique of storage in the finite control to design a TM

$$M = (Q, \{0, 1\}, \{0, 1, B\}, \delta, [q_0, B], B, [q_2, B])$$

which takes the leftmost symbol in the input and prints it immediately to the right of the input.

  - The set of states Q is $\{q_0, q_1, q_2\} \times \{0, 1, B\}$.
  - The transition function of M is as follows (a and b each may stand for either 0 or 1) :
    - $\delta([q_0, B], a) = ([q_1, a], a, R)$. Initially, q_0 is the control state, and the data portion of the state is B. The symbol scanned is copied into the second component of the state; M moves right, entering control state q_1.
    - $\delta([q_1, a], b) = ([q_1, a], b, R)$. In state q_1, M skips over each nonblank symbol and continue moving right.
    - $\delta([q_1, a], B) = ([q_2, B], a, R)$. If M reaches the first blank, it copies the symbol to from its finite control to the tape and enters the accepting state $[q_2, B]$.

Techniques for TM construction

2. Multiple Tracks :

Techniques for TM construction

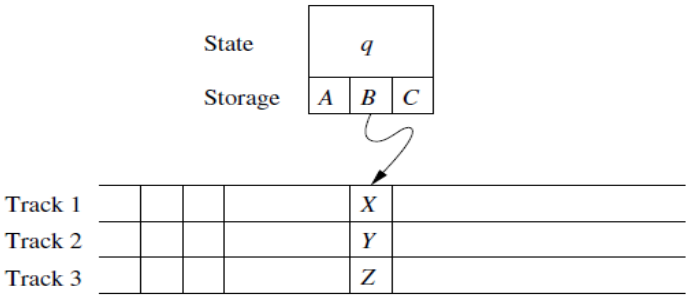
2. Multiple Tracks :

- Another useful trick is to think of the tape of a Turing machine as composed of several tracks.

Techniques for TM construction

2. Multiple Tracks :

- Another useful trick is to think of the tape of a Turing machine as composed of several tracks.



- Each track can hold one symbol, and the tape alphabet of the TM consists of tuples, with one component for each "track".
- For instance, the cell scanned by the tape head in the above figure contains the symbol [X, Y, Z].
- Like the technique of storage in the finite control, using multiple tracks does not extend what the Turing machine can do. It is simply a way to view tape symbols and to imagine that they have a useful structure.

Techniques for TM construction

- **Example :** A common use of multiple tracks is to treat one track as holding the data and a second track as holding a mark.
- We can check off each symbol as we "use" it, or we can keep track of a small number of positions within the data by marking only those positions.
- The Turing machine we shall design is

$$\text{M} = (\text{Q}, \Sigma, \Gamma, \delta, [q_0, \text{B}], [\text{B}, \text{B}], [q_4, \text{B}]) \text{ where}$$

  - Q: The set of states is $\{q_0, q_1, q_2, q_3, q_4\} \times \{0, 1, \text{B}\}$.
  - Γ : The set of tape symbols is $\{\text{B}, *\} \times \{0, 1, \text{B}\}$.
  - Σ : The input symbols are [B, 0] and [B, 1].
  - The transition function is defined by the following rules, in which a and b each may stand for either 0 or 1.
    - $\delta([q_0, B], [B, a]) = ([q_1, a], [*, a], R)$. In the initial state, M picks up the symbol a, stores it in its finite control, goes to control state q_1, "checks off" the symbol it just scanned, and moves right.
    - $\delta([q_1, a], [B, b]) = ([q_1, a], [B, b], R)$. M moves right, looking for the symbols, a blank or a checked symbol.

Techniques for TM construction

3. Subroutines :

Techniques for TM construction

3. Subroutines :

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

Techniques for TM construction

- **Example :** We shall design a TM to implement the function "multiplication". That is, our TM will start with 0^m10^n on its tape, and will end with 0^{mn} on the tape. An outline of the strategy is :
  - The tape will, in general, have one nonblank string of the form $0^i10^n10^{kn}$ for some k.
  - In one basic step, we change a 0 in the first group to B and add n 0's to the last group, giving us a string of the form $0^{i-1}10^n10^{(k+1)n}$.
  - As a result, we copy the group of n 0's to the end m times, once each time we change a 0 in the group to B. When the first group of 0's is completely changed to blanks, there will be mn 0's in the last group.
  - The final step is to change the leading 10^n1 to blanks, and we are done.

Generalized Versions of Turing Machines

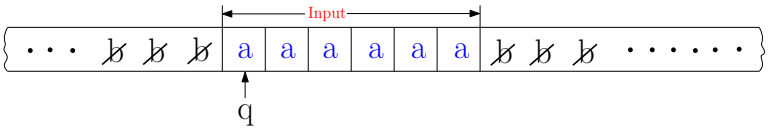
Raju Hazari

Department of Computer Science and Engineering
National Institute of Technology Calicut

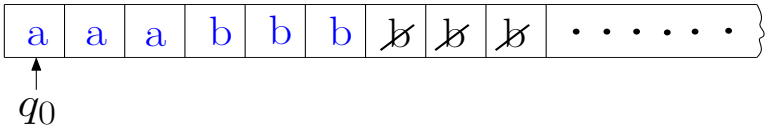
September 17-20, 2021

Two way infinite tape is equivalent to One way infinite tape

- When we looked TM as a input output device, we consider two way infinite tape.

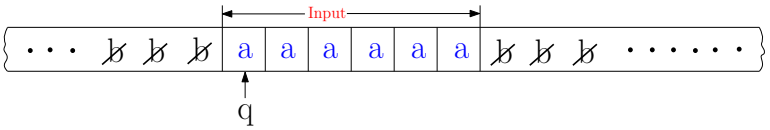


- When we looked TM as a accepting device, we consider one way infinite tape.

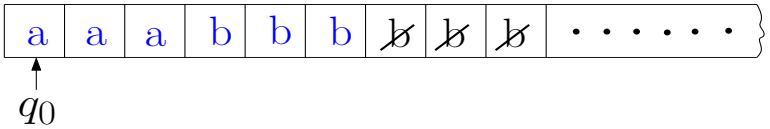


Two way infinite tape is equivalent to One way infinite tape

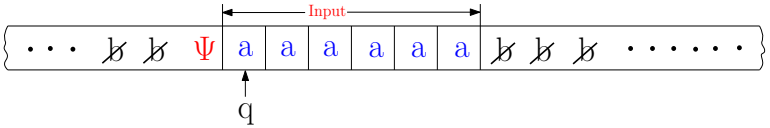
- When we looked TM as a input output device, we consider two way infinite tape.



- When we looked TM as a accepting device, we consider one way infinite tape.



- Now we can very easily simulate a one way infinite tape with two way infinite tape.
  - Use a marker in the beginning of the input

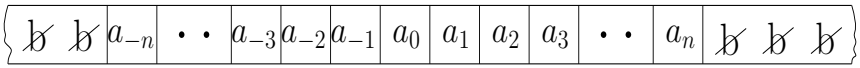


Two way infinite tape is equivalent to One way infinite tape

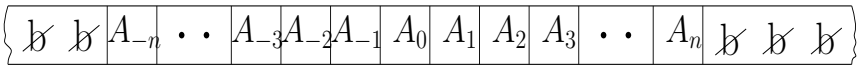
Now how do we simulate a two way infinite tape with the one way infinite tape ?

Two way infinite tape is equivalent to One way infinite tape

Now how do we simulate a two way infinite tape with the one way infinite tape ?

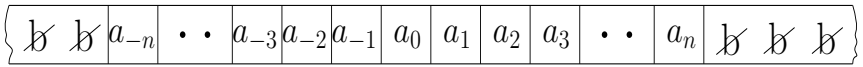


After some move

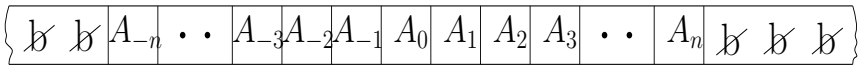


Two way infinite tape is equivalent to One way infinite tape

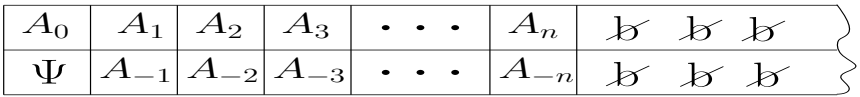
Now how do we simulate a two way infinite tape with the one way infinite tape ?



After some move



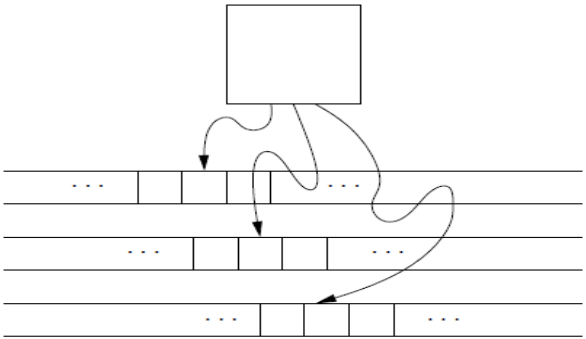
- Consider a one way infinite tape which has two track.



- So, whether you are going to read from the upper track or the lower track that information you have keep in the state.
- Therefore, if q is the state for original machine, then $[q, U]$ and $[q, L]$ are the states for this machine.
- Both the machine accepting a string with n moves.

Multitape Turing Machines

- A multitape TM has a finite control (state), and some finite number of tapes.



- Each tape is divided into cells, and each cell can hold any symbol of the finite tape alphabet.
- Initially :
  - The input is placed on the first tape.
  - All other cells of all the tapes hold the blank.
  - The finite control is in the initial state.
  - The head of the first tape is at the left end of the input.
  - All other tape heads are at some arbitrary cell.

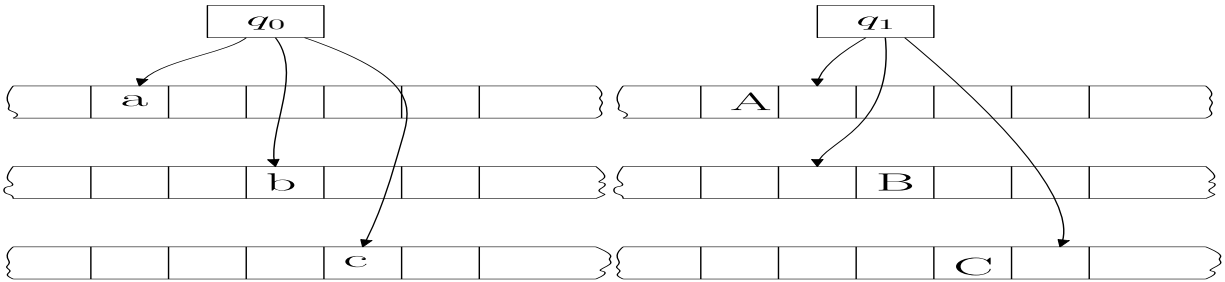
Multitape Turing Machines

- A move of the multitape TM depends on the state and the symbol scanned by each of the tape heads. In one move, the multitape TM does the following :
  - The control enters a new state, which could be the same as the previous state.
  - On each tape, a new tape symbol is written on the cell scanned. Any of these symbols may be the same as the symbol previously there.
  - Each of the tape heads makes a move, which can be either left or right. The heads move independently, so different heads may move in different directions.

Multitape Turing Machines

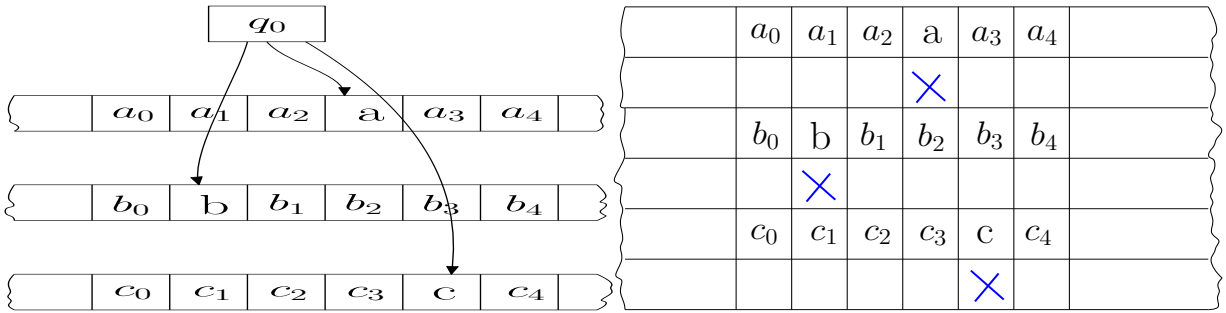
- A move of the multitape TM depends on the state and the symbol scanned by each of the tape heads. In one move, the multitape TM does the following :
  - The control enters a new state, which could be the same as the previous state.
  - On each tape, a new tape symbol is written on the cell scanned. Any of these symbols may be the same as the symbol previously there.
  - Each of the tape heads makes a move, which can be either left or right. The heads move independently, so different heads may move in different directions.

- $\delta(q_0, a, b, c) = (q_1, [A, R], [B, L], [C, R])$



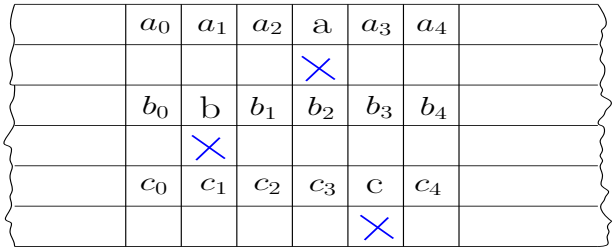
Multitape Turing Machines

- Single tape Turing machine is a particular case of a multitape Turing machine.
- We can simulate a multitape Turing machine with the single tape Turing machine.
  - If there is k tapes in a multitape TM, then we have to take a single tape TM which has $2k$ tracks.



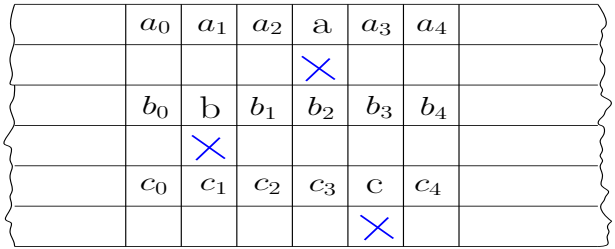
  - The odd tracks represent the contents of the tapes.
  - All the cells of even number tracks are blanks except that one cell will contain a marker, which is represent the tape head position.

Multitape Turing Machines

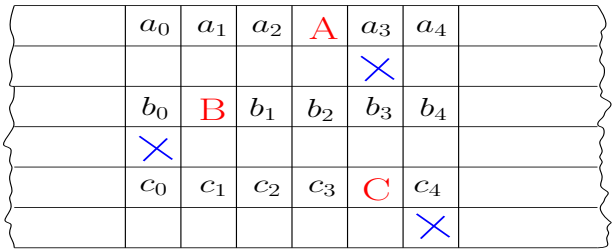


- How do you simulate one move of a multitape TM with a single tape TM

Multitape Turing Machines



- How do you simulate one move of a multitape TM with a single tape TM
  - Make one pass moving from left to right and make another pass moving from right to left.
  - $\delta(q_0, a, b, c) = (q_1, [A, R], [B, L], [C, R])$
    - ★ Pass 1 : $(q_0, 0, -, -, -)|(q_0, 1, -, b, -)|(q_0, 2, a, b, -)|(q_0, 3, a, b, c)$
    - ★ Pass 2 : $(q_0, 3, a, b, c)|(q_0, 2, a, b, -)|(q_0, 1, -, b, -)|(q_1, 0, -, -, -)$

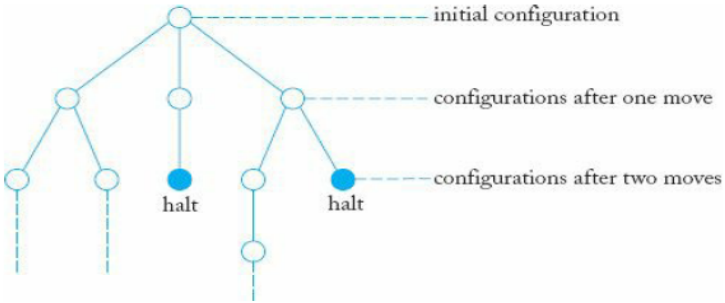


Multitape Turing Machines

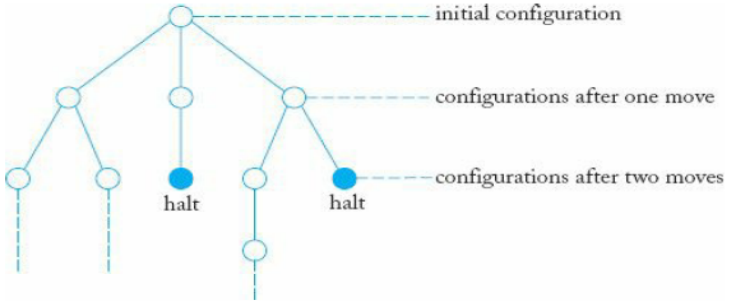
- **Theorem :** The time taken by the one-tape TM M_1 to simulate n moves of the k-tape TM M_2 is $O(n^2)$.
- **Proof :** After n moves of M_2, the tape head markers cannot have separated by more than $2n$ cells.
- Thus, if M_2 starts at the leftmost marker, it has to move no more than $2n$ cells right, to find all the head markers.
- It can then make an excursion leftward, changing the contents of the simulated tapes of M_2, and moving head markers left or right as needed.
- Doing so requires no more than $2n$ moves left, plus at most $2k$ moves to reverse direction and write a marker X in the cell.
- Thus, the number of moves by M_1 needed to simulate one of the first n moves is no more than $4n + 2k$.
- Since k is a constant, independent of the number of moves simulated, this number of moves is $O(n)$.
- To simulate n moves requires no more than n times this amount, or $O(n^2)$.

Nondeterministic Turing Machines

- A *nondeterministic* Turing machine (NTM) differ from deterministic TM by having a transition function δ such that for each state q and tape symbol X, $\delta(q, X)$ is a set of triples
$$(q_1, Y_1, D_1), (q_2, Y_2, D_2), \cdots, (q_k, Y_k, D_k)$$
where k is any finite integer.
- The NTM can choose, at each step, any of the triples to be the next move.
- It cannot, however, pick a state from one, a tape symbol from another, and the direction from yet another.
- Nondeterminism can be seen as a choice between alternatives. This can be visualized as a decision tree.

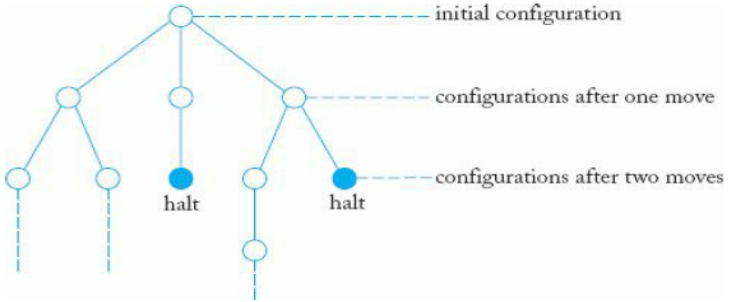


Nondeterministic Turing Machines



- The width of such a configuration tree depends on the branching factor, that is, the number of option available on each move.
- If k denotes the maximum branching, then $C = k^n$ is the maximum number of configurations that can exist after n moves.

Nondeterministic Turing Machines



- The width of such a configuration tree depends on the branching factor, that is, the number of option available on each move.
- If k denotes the maximum branching, then $C = k^n$ is the maximum number of configurations that can exist after n moves.

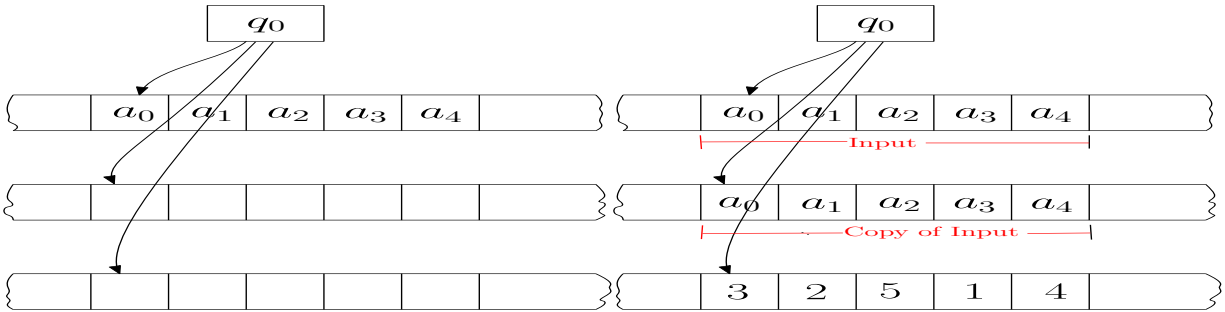
- **Definition :** A nondeterministic Turing machine M is said to *accept a language* L if, for all $w \in L$, at least one of the possible configurations accepts w.
  - There may be branches that lead to nonaccepting configurations, while some may put the machine into an infinite loop. But these are irrelevant for acceptance.

Nondeterministic Turing Machines

- **Theorem :** The class of deterministic Turing machines and the class of nondeterministic Turing machines are equivalent.

Nondeterministic Turing Machines

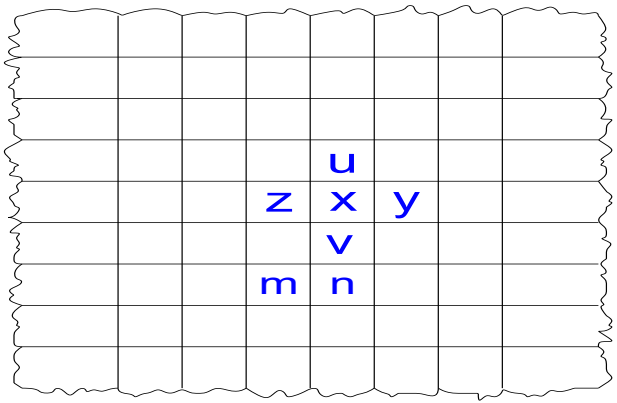
- **Theorem :** The class of deterministic Turing machines and the class of nondeterministic Turing machines are equivalent.
- **Proof :** In order to simulate a nondeterministic Turing machine with the deterministic Turing machine, the deterministic TM will have a finite control and it will have three tapes. We are taking nondeterministic TM to have one tape, you can extend the result to multitape also.



- If $\Sigma = \{1, 2, \cdots, r\}$, then the generated sequences (in standard order) $\{1, 2, \cdots r, 11, 12, \cdots, 1r, 21, 22, \cdots, 2r, 31, 32, 33, \cdots\}$
- The constructed deterministic TM may take exponentially more time than the nondeterministic TM.

Multidimensional Turing Machines

- A multidimensional Turing machine is one in which the tape can be viewed as extending infinitely in more than one dimension.



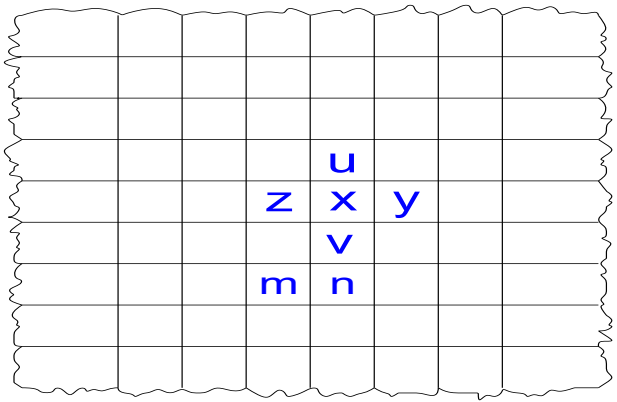
- The formal definition of a two-dimensional Turing machine involves a transition function δ of the form

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

where U and D specify movement of the read-write head up and down, respectively.

Multidimensional Turing Machines

- To simulate this machine on a standard Turing machine, we can use the two track model.

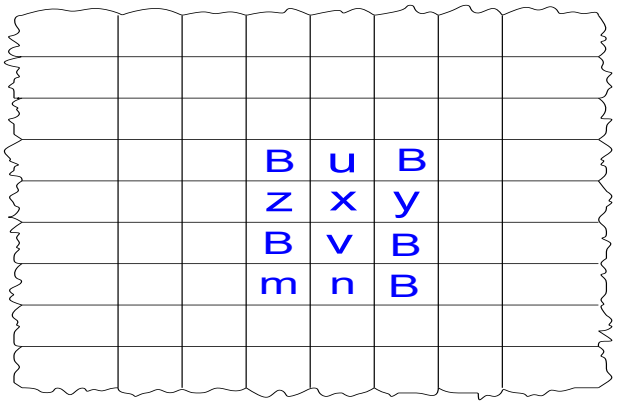


- The non-blank portion can be represented by

B u B # z x y # B v B # m n B

Multidimensional Turing Machines

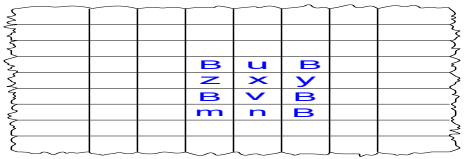
- To simulate this machine on a standard Turing machine, we can use the two track model.



- The non-blank portion can be represented by

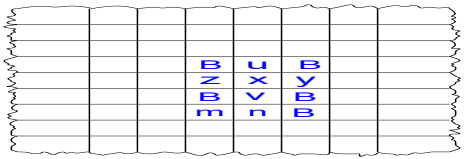
B u B # z x y # B v B # m n B

Multidimensional Turing Machines



- The non-blank portion : # B u B # z x y # B v B # m n B #
- If you want to move up, keep a count in another track and then move the next hash symbol in the left block then by the same number you move the tape head to the right.
- Similarly, if you want to move down, instead of moving left block you will move to the right block.

Multidimensional Turing Machines



- The non-blank portion : # B u B # z x y # B v B # m n B #
- If you want to move up, keep a count in another track and then move the next hash symbol in the left block then by the same number you move the tape head to the right.
- Similarly, if you want to move down, instead of moving left block you will move to the right block.
- If you want to add a row or column :
  - Add one more row up you have to create one new block on the left.
  - Add one more row down you have to create one new block on the right.
  - Add one more column to the left then in each block you introduced one blank and use the technique of shifting.
- Whatever you can do with the two dimensional tape we can do with one dimensional tape, but the number of moves will be more, but it will not be exponentially increased.

Variation of Turing Machine

- There are other variations of Turing machines :
  - Multihead Turing Machines
    - ⋆ If you have only one tape, but there may be many head.
  - Offline Turing Machines
    - ⋆ Offline Turing machine is a Turing machine where the input is given in one tape and it is read only. keep the input within marker.
    - ⋆ There is a other tape, so it is a multitape Turing machine.

Restricted Versions of Turing Machines

Raju Hazari

Department of Computer Science and Engineering
National Institute of Technology Calicut

September 27, 2021

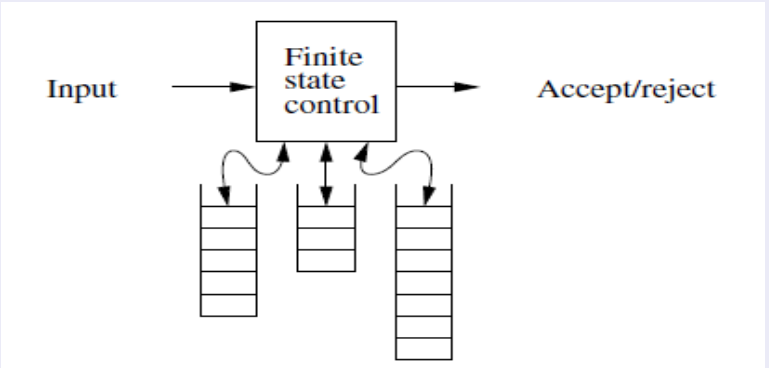
Restricted Versions of Turing Machines

- We have seen seeming generalizations of the Turing machine that do not add any language-recognizing power.
- We shall consider some restrictions on the TM that also give exactly the same language-recognizing power.
- These are the few restriction :
  - Two pushdown tape machine
  - 4 counter machine
  - 2 counter machine
  - Two symbols are enough (apart from blank symbol)
  - No rewriting

Restricted Versions of Turing Machines

Multistack Machines

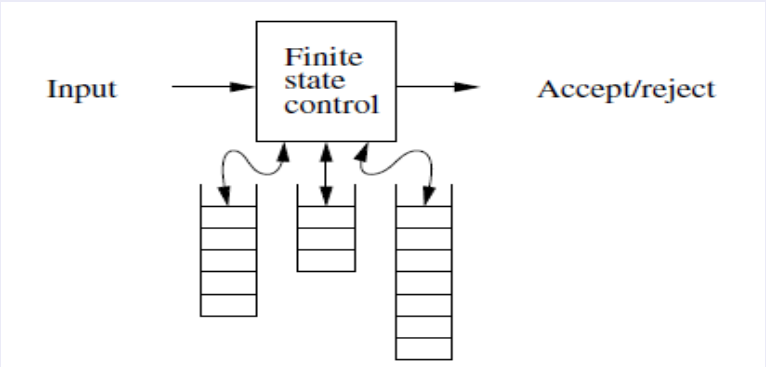
- A Turing machine can accept languages that are not accepted by any PDA (pushdown automata) with one stack.
- If we give the PDA two stacks, then it can accept any language that a Turing machine can accept.



Restricted Versions of Turing Machines

Multistack Machines

- A Turing machine can accept languages that are not accepted by any PDA (pushdown automata) with one stack.
- If we give the PDA two stacks, then it can accept any language that a Turing machine can accept.



- **Theorem** : If a language L is accepted by a Turing machine, then L is accepted by a two-stack machine.

Restricted Versions of Turing Machines

Counter Machines

- The counter machine has the same structure as the multistack machine, but in place of each stack is a counter.
- Counters hold any nonnegative integer, but we can only distinguish between zero and nonzero counters.
- In one move, the counter machine can :
  - Change state
  - Add or subtract 1 from any of its counters, independently. However, a counter is not allowed to become negative, so it cannot subtract 1 from a counter that is currently 0.
- Every language accepted by a counter machine is recursively enumerable.
- Every language accepted by a one-counter machine is a CFL (context-free language).

Restricted Versions of Turing Machines

Counter Machines

- The surprising result about counter machines is that **two counters are enough to simulate a Turing machine** and therefore to accept every recursively enumerable language.

Restricted Versions of Turing Machines

Counter Machines

- The surprising result about counter machines is that **two counters are enough to simulate a Turing machine** and therefore to accept every recursively enumerable language.

- **Theorem :** Every recursively enumerable language is accepted by a two-counter machine.

Restricted Versions of Turing Machines

Counter Machines

- The surprising result about counter machines is that **two counters are enough to simulate a Turing machine** and therefore to accept every recursively enumerable language.
- **Theorem :** Every recursively enumerable language is accepted by a two-counter machine.
- **Theorem :** Every recursively enumerable language is accepted by a four-counter machine.

Recursively Enumerable Language

Raju Hazari

Department of Computer Science and Engineering
National Institute of Technology Calicut

October 4, 2021

Turing Machine as a Generating device

- We shall consider the Turing machine as a generating device.
- We can have a Turing machine, where other tapes are also there, but there is an output tape and in this output tape strings are printed within '#' marks.

$$\# \ w_1 \ \# \ w_2 \ \# \ w_3 \ \# \cdots$$

- Some strings are printed on this tape and this tape head always moves right and something written once is not erased.
- Whatever is appearing between two hash symbols, that string is set to be generated by the Turing machine.
- If the Turing machine is denoted as M, then the language generated by Turing machine using output tape denoted by G(M).

$$\mathrm{G(M)} = \{w \mid w \text{ is generated by M}\}$$

This means w will appear between two hash symbols in the output tape and the set of all strings is denoted by G(M).

Turing Machine as a Generating device

- There is no necessity that the string should be generated any particular order.
- It is also not necessary that one string should be generated only once.
- If the strings are generated in the standard ordering manner(length wise increasing) and within the same length in the lexicographic ordering, then that language is a Recursive set.

Turing Machine as a Generating device

- There is no necessity that the string should be generated any particular order.
- It is also not necessary that one string should be generated only once.
- If the strings are generated in the standard ordering manner(length wise increasing) and within the same length in the lexicographic ordering, then that language is a Recursive set.

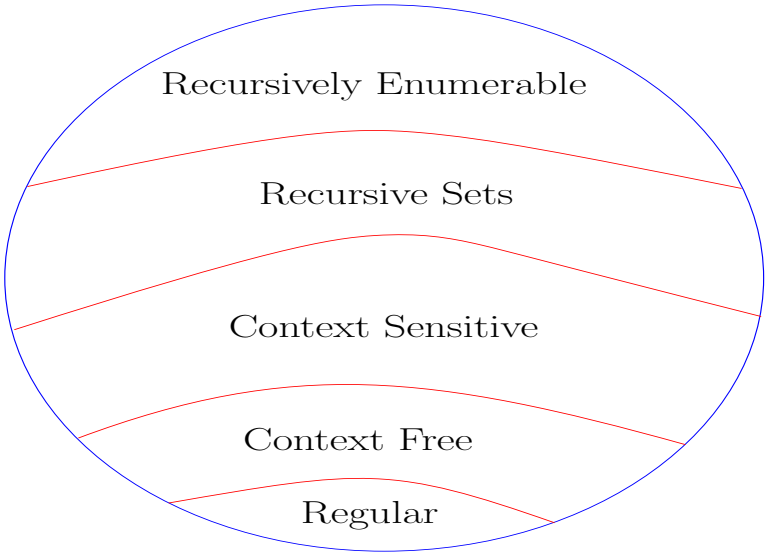
- **Theorem :** $L = L(M_1)$ for some Turing machine M_1 if and only if $L = G(M_2)$ for some Turing machine M_2.
  - If L is accepted by a Turing machine M_1 then L is generated by some other Turing machine M_2 and vice versa.

Recursively Enumerable Language

- The language accepted by a TM is called **recursively enumerable**.
- The language accepted by a TM which halts on all the inputs is called a **recursive set**.
- We seen earlier that, the language generated by a Turing machine can be compared with the language accepted by the Turing machine.
  - That is, if a language can be generated by a Turing machine in the output tape, then there is a another Turing machine which will accept it.
- So, because we are able to generate the string one by one the set is called recursively enumerable set and we know that the Turing machine need not always halt on all inputs. On some inputs it may get into a loop and never halt.
- Suppose a Turing machine halts on all inputs, then the set accepted is called a recursive set.

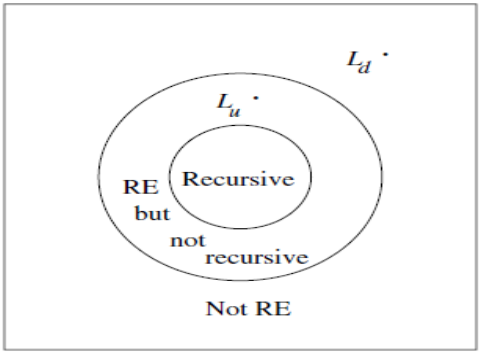
Recursively Enumerable Language

- In the Chomsky hierarchy, we have this diagram



- The recursive set form a sub class of recursively enumerable sets, but it is higher than context sensitive languages.

Relationship between the recursive, RE, and non-RE languages



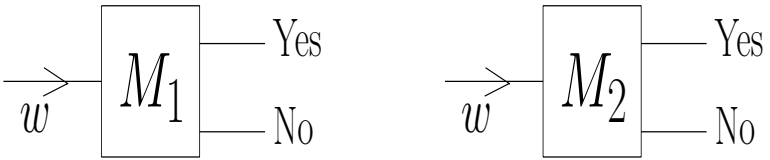
- There are three class of languages :
  - The recursive languages
  - The languages that are recursively enumerable but not recursive
  - The non-recursively-enumerable (non-RE) languages

Properties of recursive sets and recursively enumerable sets

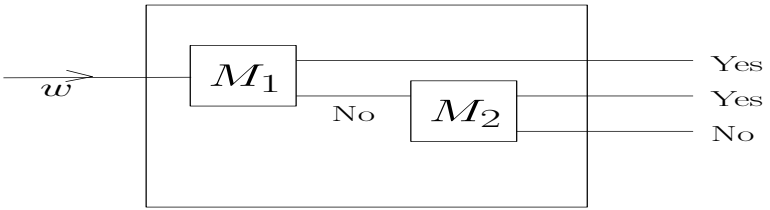
- The union of two recursive sets is recursive.
- The union of two recursively enumerable sets is recursively enumerable.
- The complement of a recursive set is recursive.
- If L and $\overline{L}$ are recursively enumerable, then L is recursive.

Properties of recursive sets and recursively enumerable sets

- The union of two recursive sets is recursive.
- **Proof :** Suppose L_1 and L_2 are recursive sets. L_1 is accept by a TM M_1 for which when input w is given. It will halt and accept or reject. Similarly, L_2 is accept by a TM M_2.



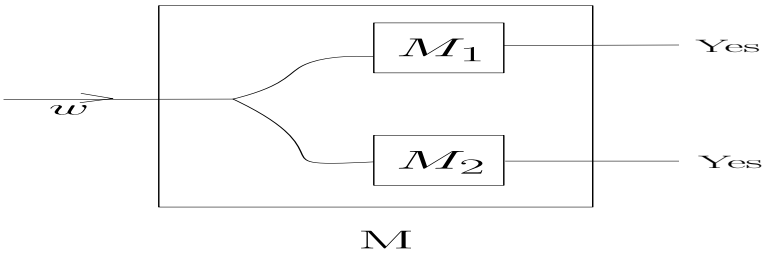
Now we can construct a compound TM M like the following structure which accept $L_1 \cup L_2$



Properties of recursive sets and recursively enumerable sets

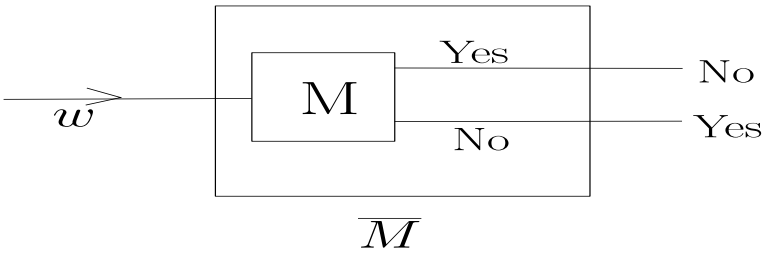
- The union of two recursively enumerable sets is recursively enumerable.
- **Proof :** Suppose L_1 and L_2 are recursively enumerable sets. L_1 is accept by a TM M_1 for which when input w is given. Similarly, L_2 is accept by a TM M_2.

Now we can construct a compound TM M like the following structure which accept $L_1 \cup L_2$



Properties of recursive sets and recursively enumerable sets

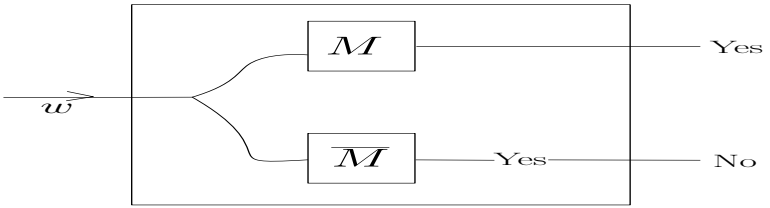
- The complement of a recursive set is recursive.
- **Proof :** Suppose L is a recursive set which is accepted by the Turing machine M. Now, we have to construct a Turing machine which always halts for $\overline{L}$.



$\overline{M}$ is the Turing machine which always halts on $\overline{L}$.

Properties of recursive sets and recursively enumerable sets

- If L and $\overline{L}$ are recursively enumerable, then L is recursive.
- **Proof :** Suppose machine for L is M and machine for $\overline{L}$ is $\overline{M}$, they need not halt on all inputs. Let us consider the composite Turing machine is M_1.



M_1

M_1 has as subparts M and $\overline{M}$ and w is given simultaneously as input to both M and $\overline{M}$. Now any string w, it has to be either in L or in $\overline{L}$. Suppose w is in L, then M say 'Yes', and if w is in $\overline{L}$ then $\overline{M}$ say 'Yes'. When $\overline{M}$ say 'Yes' then the composite machine M_1 say 'No'. So, we have a Turing machine which always halts. So, if both L and $\overline{L}$ are recursively enumerable, then we can have a Turing machine M_1 which accepting L and always halts.

That means L is a recursive set because M_1 always halts.

Properties of recursive sets and recursively enumerable sets

- If we take the two languages L and $\overline{L}$, three possibilities exist
  - Both L and $\overline{L}$ are recursive
  - Neither L nor $\overline{L}$ is recursively enumerable
  - One of L and $\overline{L}$ is recursively enumerable but not recursive ; the other is not recursively enumerable

Unrestricted Grammars

- A grammar G = (N, T, S, P) is called **unrestricted** or **Type-0** if
  - The rules are of the form: $u \to v$, where $u \in V^*NV^*$, $v \in V^*$ and $V = N \cup T$
    - u is a string of Non-Terminals and Terminals but it should have atleast one Non-Terminals.
    - If it is fully Terminal, you can not rewrite it as something else, the derivation terminate there.

Unrestricted Grammars

- A grammar G = (N, T, S, P) is called **unrestricted** or **Type-0** if
  - The rules are of the form: $u \to v$, where $u \in V^*NV^*$, $v \in V^*$ and $V = N \cup T$
    - u is a string of Non-Terminals and Terminals but it should have atleast one Non-Terminals.
    - If it is fully Terminal, you can not rewrite it as something else, the derivation terminate there.
- **Theorem 1 :** Any language generated by an unrestricted grammar is recursively enumerable.

Unrestricted Grammars

- A grammar G = (N, T, S, P) is called **unrestricted** or **Type-0** if
  - The rules are of the form: $u \rightarrow v$, where $u \in V^*NV^*$, $v \in V^*$ and $V = N \cup T$
    - u is a string of Non-Terminals and Terminals but it should have atleast one Non-Terminals.
    - If it is fully Terminal, you can not rewrite it as something else, the derivation terminate there.

- **Theorem 1** : Any language generated by an unrestricted grammar is recursively enumerable.

- **Theorem 2** : For every recursively enumerable language L, there exits an unrestricted grammar G, such that L = L(G).

Halting Problem

Raju Hazari

Department of Computer Science and Engineering
National Institute of Technology Calicut

October 29, 2021

Encoding of a Turing machine

- A Turing machine can be encoded as a binary string.
- Let us consider a Turing machine $M = (K, \Sigma, \Gamma, \delta, q_1, F)$ where $K = \{q_1, \cdots, q_n\}$, $\Sigma = \{0, 1\}$, $\Gamma = \{0, 1, B\}$, q_1 : initial state, and the mapping will be form $\delta(q_i, X_j) = (q_k, X_l, L/R)$
- We can have three symbol, we may consider $X_1 = 0$, $X_2 = 1$, $X_3 = B$.
- The value of j and l is should be between 1 and 3. The value of i and k is should be between 1 and n.
- The move can be left or right. So, if it is left we can use 1, if it is right we can use 2.
- Now, the move $\delta(q_i, X_j) = (q_k, X_l, L/R)$ can be encoded as a binary string like this : $0^i10^j10^k10^l10^m$, $m = 1 = D_1$ if it is left move, $m = 2 = D_2$ if it is right move.
- Any move of the Turing machine can be encoded in binary like this.

Encoding of a Turing machine

- Let TM $M = (\{q_1, q_2, q_3\}, \{0, 1\}, \{0, 1, B\}, \delta, q_1, B, \{q_2\})$ where δ consists of the rules :

$$\delta(q_1, 1) = (q_3, 0, R)$$
$$\delta(q_3, 0) = (q_1, 1, R)$$
$$\delta(q_3, 1) = (q_2, 0, R)$$
$$\delta(q_3, B) = (q_3, 1, L)$$

- The codes for each of these rules, respectively, are :

$$m_1 : 0100100010100$$
$$m_2 : 0001010100100$$
$$m_3 : 00010010010100$$
$$m_4 : 0001000100010010$$

- For example, the first rule can be written as $\delta(q_1, X_2) = (q_3, X_1, D_2)$, since $1 = X_2$, $0 = X_1$, and $R = D_2$. Thus its code is $0^1 10^2 10^3 10^1 10^2$

- The encoding of the Turing machine like this $111m_111m_211m_311m_4111$. Each move separated by two one's. You could have arranged them in a different order also, that also represents the same Turing machine. It will be a different encoding. Now, in the beginning, you have three one's and in the ends you have three one's. So, the encoding of a Turing machine begins with three one's and ends with three one's.

- A code for M is :

$$111010010001010011000101010010011000100100101001100010001000100010010111$$

Decidability

- The concept of "decidability" is a breakthrough in Theoretical Computer Science and Mathematics in the first half of 20^{th} century. After this, it become clear that many of the known problems such as Hilbert's 10^{th} problem are undecidable.
- This concept was stated by Turing in 1936, and two of the results have been considered as very important in Theoretical Computer Science.
- One is the *Gödel*'**s incompleteness theorem** and other is the **halting problem** for the Turing machine.
- Till this result was proved people were trying on so many problems, trying to develop algorithms and so on.

Decidability

- But after this result was proved, they knew that for some problems no body can ever write an algorithm. Hilbert's 10^{th} problem was on of them.
  - Hilbert's 10^{th} problem : In 1901, Hilbert stated this problem, that is, "can you write an algorithm which will take as input a polynomial with integer coefficients and that equation polynomial equal to zero, whether that has integer roots or not".

Decidability

- But after this result was proved, they knew that for some problems no body can ever write an algorithm. Hilbert's 10^{th} problem was on of them.
  - Hilbert's 10^{th} problem : In 1901, Hilbert stated this problem, that is, "can you write an algorithm which will take as input a polynomial with integer coefficients and that equation polynomial equal to zero, whether that has integer roots or not".
- That also gave an idea about the Fermat's last theorem.
  - Fermat's last theorem : "Can you find integers x, y, z such that $x^n + y^n = z^n$ for $n \geq 3$, and his conjecture was that you cannot find integers"
  - For a long time it was not proved and finally few year (15-20 year) back it was proved. The result that the halting problem is undecidable gave a clue to this.

Halting Problem for Turing Machine

- The concept of Undecidability was introduced by showing that the halting problem for Turing machine is undecidable.
- The halting problem for Turing machines can be stated as follows : "Given a Turing machine in an arbitrary configuration will it eventually halt ?"
  - That means, can you give an algorithm which will take a Turing machine and an input as input and say whether the Turing machine on that input will halt or not ?
- This problem is said to be recursively unsolvable or undecidable in the sense that there cannot exist an algorithm which will take as input a description of a Turing machine T and an input t and say whether T on t will halt or not.

Halting Problem for Turing Machine is Undecidable

- **Proof :** Suppose the halting problem is decidable. Then there should be an algorithm to solve it and a corresponding Turing machine M_1. This machine M_1 take as input an encoding d_T of a TM or description of TM and an input t for the Turing machine and tells whether T on t halt or not. The state diagram may be given as follows :

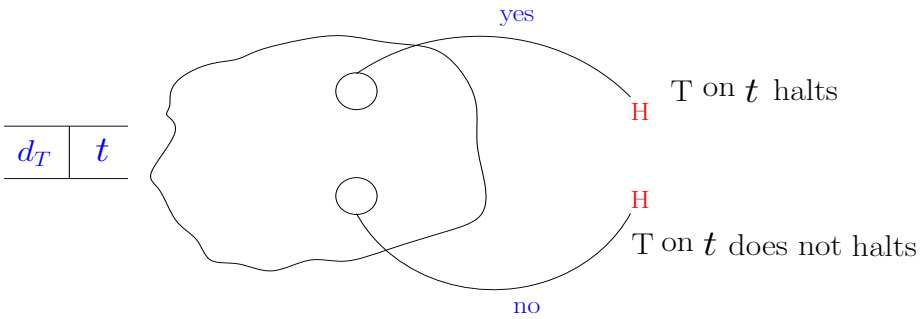


Fig : Machine M_1

- So given this input, if T on t halts the machine will go to a state, take exit and halt and say yes.
- Suppose T on t does not halt , then the machine will go to a state, take exit, say no and halt.

Halting Problem for Turing Machine is Undecidable

- The machine M_1 can be modified a little and we can think of a Turing machine M_2 as follows :

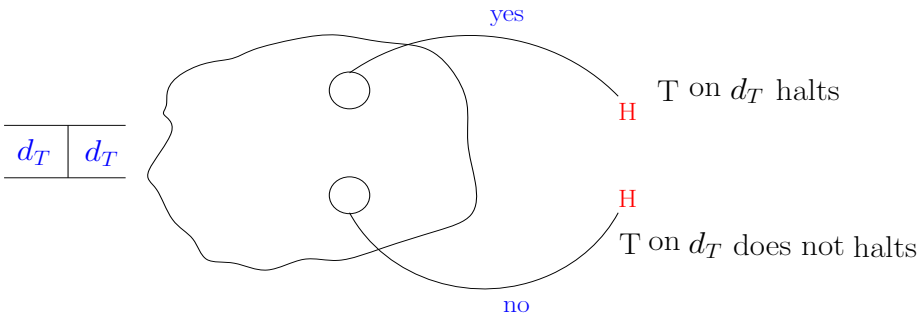


Fig : Machine M_2

- We know that the encoding of a Turing machine is a binary string, and the input is also a binary string. So, instead of taking any t, we take t as the encoding itself.
- That is, the machine starts with one binary string d_T, then it makes a copy of that string, then call M_1 as a subroutine.
- So, it will answer the question whether the machine T taking its own encoding will halt or not.
- If the machine T halts on the binary string (d_T), it will take 'yes' exit and halt. If the machine T does not halt on this binary string it will take 'no' exit and halt.
- So, if it is possible to have a machine like M_1, it is possible to have a machine like M_2.

Halting Problem for Turing Machine is Undecidable

- Now we can modify M_2 a little and have a machine M_3 as follows :

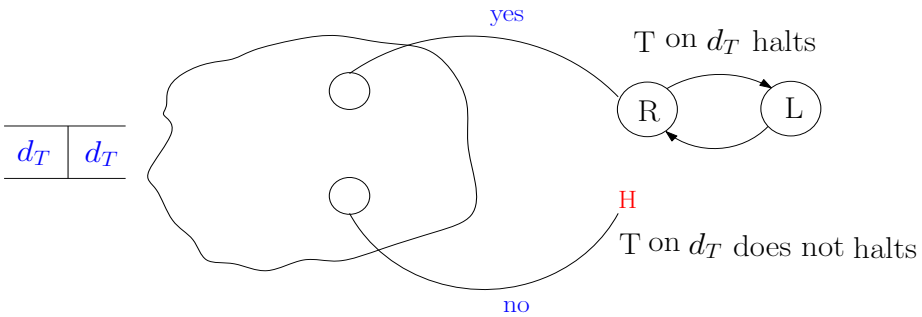


Fig : Machine M_3

- The machine M_3 is almost like M_2 except that when the 'yes' exit is taken, the machine oscillates between two states and moves left and right between two consecutive cells.

- Suppose we give as input to M_3 the encoding of M_3. It will try to find whether M_3 will halt on its own encoding. If it halts. it has to take 'yes' exit and will get into a loop. If it does not halt, it will take 'no' exit and will halt. Hence we arrive at a contradiction.

- Hence it is not possible to have a Turing machine like M_3 and also M_2 and M_1. Hence, the halting problem for Turing machine is undecidable.

Decidability

Raju Hazari

Department of Computer Science and Engineering
National Institute of Technology Calicut

November 1, 2021

Decidability

Problem and Instance of a Problem

- Informally problem refers to a question such as “Is the language accepted by a Turing machine empty ?” or something like “Does a graph have a Hamiltonian circuit ?”.
- (For first problem) The instance of a problem is a particular Turing machine. For, Hamiltonian graph problem, the instance is a graph.
- We can restrict ourselves to yes/no problem called **decision problems**. These problems can be encoded as strings and we can think of a Turing Machine which will accept those strings which encode ‘yes’ instances and reject those strings which encode ‘no’ instances of the problem.

Problems, Instances and Languages

- A decision problem is a problem for which the answer is 'Yes' or 'No'.
  - If you can design an algorithm which will come out with an answer 'yes' or 'no' then the problem is decidable.
  - If you cannot have an algorithm then the problem is undecidable.
- For example, **satisfiability problem** means to find out whether a Boolean expression is satisfiable or not. We call this as SAT.
  - The Boolean expression is like this : $(x_1 + x_2)(\neg x_2 + x_3 + x_4) + x_1 x_4$
  - This Boolean expression is satisfiable or not means, does there existence assignments (to the variable $x_1, x_2, \cdots$, etc 0 or 1) which will make the expression evaluate to 1 or True. So, if it evaluates to 1, some assignment makes the expression evaluate to 1, then you say that instance has a solution. So, that expression is satisfiable; that particular instance is satisfiable.
  - There are instances which are not satisfiable. For example : $p \wedge \neg p$, $(x_1)(\neg x_1)$. These two expression are never satisfied.

Problems, Instances and Languages

- Also AMB is ambiguity problem for CFG– i.e., finding out whether a CFG is ambiguous or not.
  - If any string will have only one derivation tree, the grammar is unambiguous. For example : $G_1 : S \rightarrow aSb, S \rightarrow ab$.
  - If any string will have more than one derivation tree, the grammar is ambiguous. For example : $G_2 : S \rightarrow SS, S \rightarrow a$.
- A particular CFG is an instance of the problem. If this particular CFG is ambiguous, it is an 'yes' instance of the problem. If it is not ambiguous it is 'no' instance of the problem.
- We know that G_1 is unambiguous grammar and G_2 is ambiguous grammar. These are particular instance of the problem. For particular instance you may be able to find out the solution. When you say that the problem is undecidable, what is means is, in general, there is no algorithm which will take a context free grammar G as input and come out with 'yes' it is ambiguous or 'no' it is unambiguous.

Problems, Instances and Languages

- Similarly, a particular Boolean expression is an instance of SAT problem. If it is satisfiable it is an 'yes' instance of the problem. If it is not satisfiable it is a 'no' instance of the problem.
- In contrast to these problems, problem is like finding a Hamiltonian circuit in graph are called optimization problems. Whether a graph has a Hamiltonian circuit or not is decision problem. Finding one is an optimization problem.

Problems, Instances and Languages

- Any instance of a problem can be encoded as a string.
- A Boolean expression can be looked at as a string. For example $(x_1 + x_2)(\overline{x_1} + \overline{x_2})$ can be written as $(x1 + x2)(\neg x1 + \neg x2)$ – a string from the alphabet $\{(,), x, +, \neg, 0, 1, 2, \cdots, 9\}$.
- A CFG $G = (N, T, P, S)$ can be looked at string over $N \cup T \cup \{(,), \{, \}, \rightarrow, \}$. For example, the grammar $(\{S\}, \{a, b\}, P, S)$ where P consists of $S \rightarrow aSb$, $S \rightarrow ab$ can be looked at as a string $(\{S\}, \{a, b\}\{S \rightarrow aSb, S \rightarrow ab\}, S)$.
- It is to be noted that generally integers are represented as decimal or binary and not as unary. The problem can be re-formulated as one recognizing the language consisting of the 'yes' instances of the problem.
- So, any problem we can encode as a binary string. So, this is the connection between problems and languages. When we say that the problem is undecidable, it is equivalent to saying that the corresponding languages which consists of the 'yes' instances is not recursive.

Decidability

- After the halting problem for Turing machine was proved to be undecidable, many problems for which researches were trying to find algorithms were shown to be undecidable.

Decidability

- After the halting problem for Turing machine was proved to be undecidable, many problems for which researches were trying to find algorithms were shown to be undecidable.

- How we proved that a new problem is undecidable ?

- **Proof :** For proving the new problem undecidable, a known undecidable problem has to be reduced to it. One instance of the known problem has to be converted into one instance of the new problem.
  Let P_{new} be the problem which is to be shown to be undecidable and let P_{known} be the known undecidable problem. There will be several instances in both P_{new} and P_{known}, but there should be a way of converting one instance of P_{known} to one instance of P_{new}. So, while constructing one instance of P_{new} from one instance of P_{known}, we have to convert 'yes' instances into 'yes' instances and 'no' instances into 'no' instances.
  The argument will be like this– Suppose P_{new} were decidable, that means there exist an algorithm (Let A_1) for that. So, we can take one instance of the P_{known} problem, convert it into an instance of P_{new} problem and call the algorithm A_1. This algorithm will solve it. So, there is a solution for the P_{known} also. So, we can say that P_{known} is decidable. But we know that P_{known} is undecidable. So, we arriving at a contradiction. So, that shows P_{new} is not decidable.
  We call this as reducing P_{known} to P_{new}, or sometimes $P_{known} \sim P_{new}$ (P_{known} is reduced to P_{new}) and show that P_{new} is undecidable.

Decidability

- Many proves can be proved to be undecidable by reducing the halting problem to it. But for problems related to formal languages theory, like ambiguity problem, we rather used another problem called post correspondence problem (PCP). So, first we prove that PCP is undecidable from the halting problem, then making use of this PCP we show that other problems are undecidable.

Universal Turing Machine

- **Universal Turing machine** is TM which can simulate any other TM including itself.

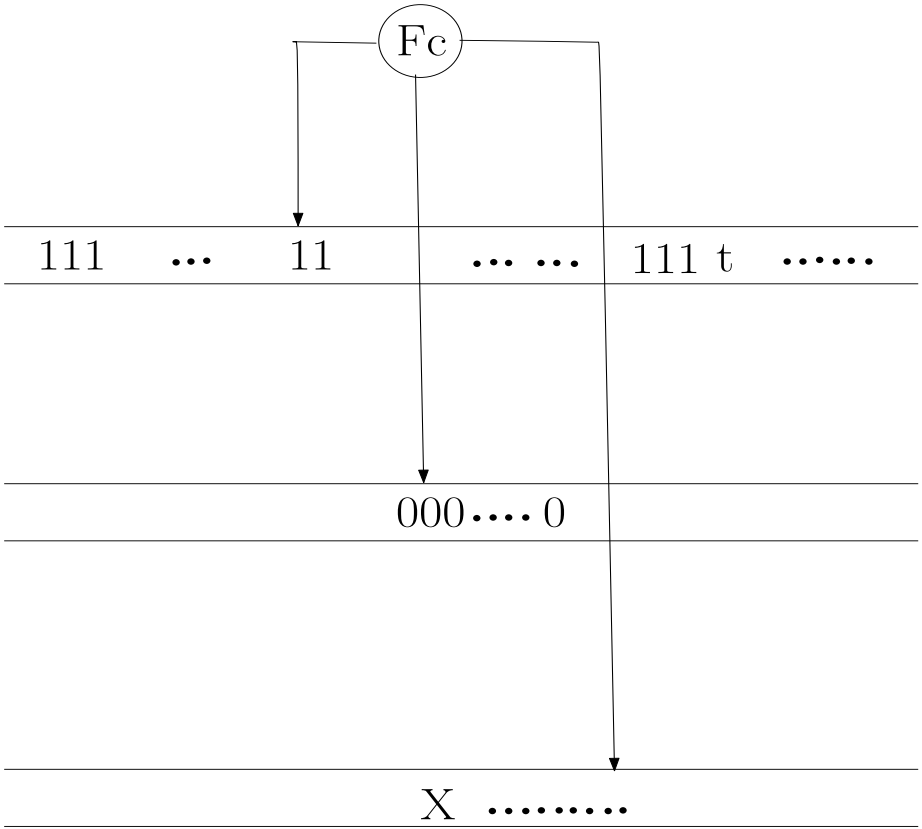
- Let us denote a Universal TM by U. Without loss of generality, we are considering TMs with tape alphabets $\{0, 1, B\}$. The encoding of such TM T is a binary string d_T. U has three tapes. The first tape is represented with d_T t ; i.e., the encoding of T and the input t to T. It will be of the form :

111	...		11	...	...		111	t	.....

- Note that t is a string over $\{0, 1\}$. The second tape initially has a 0. Without loss of generality, we can assume that T states $\{q_1, q_2, \cdots, q_k\}$ where q_1 is the initial state and q_2 is the final state. The second tape contains the information about the state. At any instance T is supposed to be in state q_i, U while simulating T will have 0^i in tape 2. Tape 3 is used for simulation.

Universal Turing Machine

- U can be represented by the following figure :



Universal Turing Machine

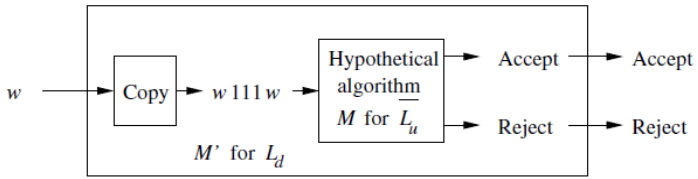
- Initially, second tape contains single 0, which means state 0. At any time, it may have some 0's which will denotes the states. The third tape is used for simulation and initially t is copied in this tape. The Universal Turing machine before doing this, it checks whether it is a proper encoding or not. If it represent improper encoding, the machine halts. A machine with improper encoding represents a Turing machine with no moves and so it cannot accept any string.
  Now, once it find it is a proper encoding, t is copied on to the third tape and the behavior of this Turing machine is simulated on the third tape. Suppose, in the third tape a binary string like 0110. The third tape head reading a 1 and consider the contains of second tape is four 0's, that means in state 4. It is reading a 1, then it goes through the first tape and check whether there is a block beginning $0^4 10^2 1 \cdots$. If there is a block which begin like this, there is a move for this situation. Then it stores in its memory what is given there something like $0^k 10^l 10^m$. So, it replace the contain of second tape with 0^k and print in the third tape 0 or 1 depending upon the value of l and then depending upon the value of m, tape head position in tape three will moved left or right. In this way, we simulate one move of the Turing machine in Universal Turing machine. So, if at some time 00 appears in the second tape, that means you have reached the final state. So, this string will be accepted.

Universal Language

- We define the *universal language* L_u to be the set of binary strings that encode a pair (M, w), where M is a TM with the binary input alphabet, and w is a siting in $(0+1)^*$, such that w is in $L(M)$.
- That is, L_u is the set of strings representing a TM and an input accepted by that TM.

Undecidability of the Universal Language

- **Theorem :** L_u is RE but not recursive.
- **Proof :** We know that L_u is RE. Suppose L_u were recursive, then $\overline{L_u}$, the complement of L_u would also be recursive. However, if we have a TM M to accept $\overline{L_u}$, then we can construct a TM to accept L_d (diagonalization language). Since we already know that L_d is not RE, we have a contradiction of our assumption that L_u is recursive.



Reduction of L_d to $\overline{L_u}$

- Suppose $L(M) = \overline{L_u}$. We can modify TM M into a TM $M^{'}$ that accepts L_d as follows.
  - Given string w on its input, $M^{'}$ changes the input to $w111w$.
  - $M^{'}$ simulates M on the new input. If w is w_i, then $M^{'}$ determines whether M_i accepts w_i. Since M accepts $\overline{L_u}$, it will accept if and only if M_i does not accept w_i; i.e., w_i is in L_d.
- Thus, $M^{'}$ accepts w if and only if w is in L_d. Since we know $M^{'}$ cannot exit, we conclude that L_u is not recursive.

Undecidable Problems about Turing Machines

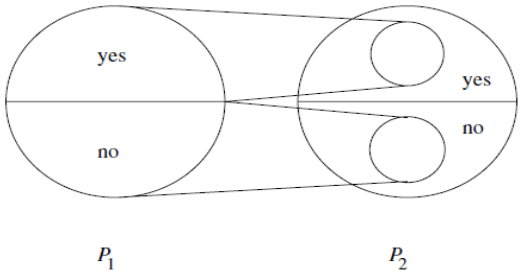
Raju Hazari

Department of Computer Science and Engineering
National Institute of Technology Calicut

November 8-10, 2021

Reductions

In general, if we have an algorithm to convert instances of a problem P_1 to instances of a problem P_2 that have the same answer, then we say that P_1 ***reduces*** to P_2.



Reductions turn positive instances into positive, and negative instance to negative

Theorem : If there is a reduction from P_1 to P_2, then :

(a) If P_1 is undecidable then so is P_2.

(b) If P_1 is non-RE, then so is P_2.

Turing Machines that accept the Empty Language

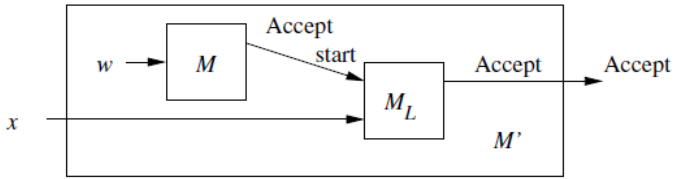
- Let us consider two language L_e and L_{ne}. Each consists of binary strings.
- If w is a binary string, then it represents some TM, M_i.
- If $L(M_i) = \emptyset$, that is, M_i does not accept any input, then w is in L_e. Thus, L_e is the language consisting of all those encoded TM's whose language is empty.
  - $L_e = \{M \mid L(M) = \emptyset\}$
- On the other hand, if $L(M_i)$ is not the empty language, then w is in L_{ne}. Thus, L_{ne} is the language of all codes for Turing machines that accept at least one input string.
  - $L_{ne} = \{M \mid L(M) \neq \emptyset\}$
- L_e and L_{ne} are both languages over the binary alphabet $\{0, 1\}$, and that they are complements of one another.
- **Theorem :** L_{ne} is recursively enumerable, but not recursive. L_e is not recursively enumerable.

Rice's Theorem and Properties of the RE Languages

Let S be a set of recursively enumerable languages, each a subset of $(0+1)^*$. A set L has a property S if $L \in S$. S is a trivial property if S is empty or S consists of all recursively enumerable languages.
Let $L_S = \{<\text{M}>|\text{L(M) is in S}\}$.

Theorem : Any non-trivial property S of recursively enumerable languages is undecidable.

Proof : Let S be a nontrivial property of the RE languages. Assume that $\emptyset$, the empty language, is not in S. Since S is nontrivial, there must be some nonempty language L that is in S. Let M_L be a TM accepting L.
We shall reduce L_u to L_S, thus proving that L_S is undecidable, since L_u is undecidable. The algorithm to perform the reduction takes as input a pair (M, w) and produces a TM $M^{'}$.



$L(M^{'})$ is $\emptyset$ if M does not accept w, and $L(M^{'}) = L$ if M accepts w. $M^{'}$ is a two-tape TM. One tape is used to simulate M on w. The other tape of $M^{'}$ is used to simulate M_L on the input x to $M^{'}$, if necessary. Again, the transitions of M_L are known to the reduction algorithm and may be "build into" the transitions of $M^{'}$.

Rice's Theorem and Properties of the RE Languages

The TM $M^{'}$ is constructed to do the following :

- Simulate M on input w. Note that w is not the input to $M^{'}$; rather, $M^{'}$ writes M and w onto one of its tapes and simulates the universal TM U on that pair.
- If M does not accept w, then $M^{'}$ does nothing else. $M^{'}$ never accepts its own input x, so $L(M^{'}) = \emptyset$. Since we assume $\emptyset$ is not in property S, that means the code for $M^{'}$ is not in L_S.
- If M accepts w, then $M^{'}$ begins simulating M_L on its own input x. Thus, $M^{'}$ will accept exactly the language L. Since L is in S, the code for $M^{'}$ is in L_S.

We observe that constructing $M^{'}$ from M and w can be carried out by an algorithm. Since this algorithm turns (M, w) into an $M^{'}$ that is in L_S if and only if (M, w) is in L_u, this algorithm is a reduction of L_u to L_S, and proves that the property S is undecidable.

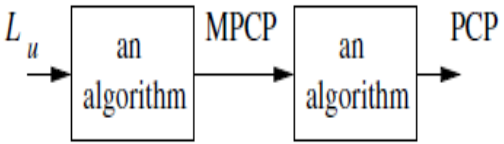
Problems about Turing Machine Specifications

The following problems are undecidable :

- Whether the language accepts by a TM is empty.
- Whether the language accepted by a TM is finite.
- Whether the language accepted by a TM is a regular language.
- Whether the language accepted by a TM is a context-free language.

Post's Correspondence Problem (PCP)

- **Theorem :** Post's Correspondence Problem is undecidable.
- We can prove PCP undecidable by reducing L_u to PCP.
- To facilitate the proof we introduce a "modified" PCP, and reduce the modified problem to the original PCP. Then we reduce L_u to the modified PCP.
- The chain of reductions is suggested in the below figure.



- Since the original L_u is known to be undecidable, we conclude that PCP is undecidable.

Post's Correspondence Problem (PCP)

- An instance of Post's Correspondence Problem (PCP) consists of two lists of strings over some alphabet Σ; the two lists must be of equal length.
- We generally refer to the A and B lists, and write $A = w_1, w_2, \cdots, w_k$ and $B = x_1, x_2, \cdots, x_k$, for some integer k.
- For each i, the pair (w_i, x_i) is said to be a *corresponding* pair.
- We say this instance of PCP has a *solution*, if there is a sequence of one or more integers $i_1, i_2, \cdots, i_m$ that, when interpreted as indexes for strings in the A and B lists, yield the same string. That is, $w_{i_1} w_{i_2} \cdots w_{i_m} = x_{i_1} x_{i_2} \cdots x_{i_m}$.
- We say the sequence $i_1, i_2, \cdots, i_m$ is a solution to this instance of PCP, if so.

Post's Correspondence Problem (PCP)

- An instance of Post's Correspondence Problem (PCP) consists of two lists of strings over some alphabet Σ; the two lists must be of equal length.
- We generally refer to the A and B lists, and write $A = w_1, w_2, \cdots, w_k$ and $B = x_1, x_2, \cdots, x_k$, for some integer k.
- For each i, the pair (w_i, x_i) is said to be a *corresponding* pair.
- We say this instance of PCP has a *solution*, if there is a sequence of one or more integers $i_1, i_2, \cdots, i_m$ that, when interpreted as indexes for strings in the A and B lists, yield the same string. That is, $w_{i_1} w_{i_2} \cdots w_{i_m} = x_{i_1} x_{i_2} \cdots x_{i_m}$.
- We say the sequence $i_1, i_2, \cdots, i_m$ is a solution to this instance of PCP, if so.
- The Post's correspondence problem is :
  - Given an instance of PCP, tell whether this instance has a solution.

Post's Correspondence Problem (PCP)

- **Example :** Let $\Sigma = \{0, 1\}$, and let A and B lists be as defined in Table

	List A	List B
i	w_i	x_i
1	110	110110
2	0011	00
3	0110	110

- In this case, PCP has a solution. For instance, let $m = 3$, $i_1 = 2$, $i_2 = 3$ and $i_3 = 1$, i.e., the solution is the list 2, 3, 1.

- We verify that this list is a solution by concatenating the corresponding strings in order for the two lists, that is,

$$w_2 w_3 w_1 = x_2 x_3 x_1 = 00110110110$$

- This solution is not unique. For instance, 2, 1, 1, 3, 2, 1, 1, 3 is another solution.

Post's Correspondence Problem (PCP)

- **Example :** Let $\Sigma = \{0, 1\}$, and let A and B lists be as defined in Table

	List A	List B
i	w_i	x_i
1	011	101
2	11	011
3	1101	110

- In this case, PCP has no solution.

Modified Post's Correspondence Problem (MPCP)

- In the modified PCP, there is the additional requirement on a solution that the first pair on the A and B lists must be the first pair in the solution.

- More formally, an instance of MPCP is two lists $A = w_1, w_2, \cdots, w_k$ and $B = x_1, x_2, \cdots, x_k$, and a solution is a list of 0 or more integers $i_1, i_2, \cdots, i_m$ such that

$$w_1 w_{i_1} w_{i_2} \cdots w_{i_m} = x_1 x_{i_1} x_{i_2} \cdots x_{i_m}$$

- The pair (w_1, x_1) is forced to be at the beginning of the two strings, even though the index 1 is not mentioned at the front of the list that is the solution.

- Unlike PCP, where the solution has to have at least one integer on the solution list, in MPCP, the empty list could be a solution if $w_1 = x_1$.

Modified Post's Correspondence Problem (MPCP)

- **Example :** Let $\Sigma = \{0, 1\}$, and let A and B lists be as defined in Table

	List A	List B
i	w_i	x_i
1	110	110110
2	0011	00
3	0110	110

- However, as an instance of MPCP it has no solution.

- In proof, observe that any partial solution has to begin with index 1, so the two strings of a solution would begin.

$$A : 110 \cdots$$
$$B : 110110 \cdots$$

- The next integer could not be 2 or 3, since both w and w begin with 0 and thus would produce a mismatch at the fourth position. Thus, the next index would again have to be 1, yielding

$$A : 110110 \cdots$$
$$B : 110110110110 \cdots$$

- We can argue this way indefinitely. Once again only another 1 in the solution can avoid a mismatch, but if we can only pick index 1, the B string remains twice as long as the A string, and the two strings can never become equal.

Undecidability of Ambiguity for CFG's

- **Theorem :** It is undecidable whether a CFG is ambiguous.
- **Theorem :** Let G_1 and G_2 be context-free grammars, and let R be a regular expression. Then the following are undecidable :
  - Is $L(G_1) \cap L(G_2) = \emptyset$?
  - Is $L(G_1) = L(G_2)$?
  - Is $L(G_1) = L(R)$?
  - Is $L(G_1) = T^*$ for some alphabet T ?
  - Is $L(G_1) \subseteq L(G_2)$?
  - Is $L(R) \subseteq L(G_1)$?

Complexity Theory

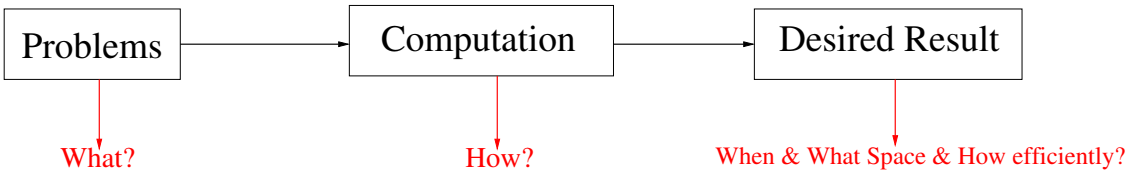
Raju Hazari

Department of Computer Science and Engineering
National Institute of Technology Calicut

November 10-15, 2021

Theory of Computation

1. What kind of problem can be computed ?
2. How a problem can be computed ?
   - Means, what is the method by which a problem can be computed to achieve the desired result.
3. When & What Space & How Efficiently can a problem can be computed ?



- All the 3 questions will be answer in Theory of Computation, which means the field Theory of Computation is divided into three parts.
  1. "What" kind of problem can be computed, which means COMPUTABILITY THEORY.
  2. "How" a problem can be computed, which is AUTOMATA THEORY.
  3. When & What Space & How Efficiently can a problem can be computed, means it is COMPLEXITY THEORY.

Complexity Theory

Intractable Problems

- From a computational complexity stance, ***intractable problems*** are problems for which there exist no efficient algorithms to solve them.
- Most intractable problems have an algorithm – the same algorithm – that provides a solution, and that algorithm is the brute-force search.
- This algorithm, however, does not provide an efficient solution and, therefore, not feasible for computation with anything more than the smallest input.
- Example of Intractable Problems:
  - Factoring a number into primes.
  - Integer partition: Can you partition n integers into two subsets such that the sums of the subsets are equal?

- Even when a problem is decidable and thus computationally solvable in principle, it may not be solvable in practice if the solution requires an inordinate amount of time or memory.

Time and Space Complexity

Time Complexity

- Consider multitape Turing machine M with k infinite tapes. If for every input word of length n, M makes at most $T(n)$ moves before halting, then M is said to be a $T(n)$ time bounded Turing machine or of time complexity $T(n)$. The language recognized by M is said to be of time complexity $T(n)$.

Time and Space Complexity

Time Complexity

- Consider multitape Turing machine M with k infinite tapes. If for every input word of length n, M makes at most $T(n)$ moves before halting, then M is said to be a $T(n)$ time bounded Turing machine or of time complexity $T(n)$. The language recognized by M is said to be of time complexity $T(n)$.

Space Complexity

- Consider the offline Turing machine M. If for every input word of length n, M scans at most $S(n)$ cells on any storage tape, then M is said to be an $S(n)$ space bounded Turing machine or of space complexity $S(n)$. The language recognized by M is also said to be of space complexity $S(n)$.

The Classes $\mathcal{P}$ and $\mathcal{NP}$

- The Classes $\mathcal{P}$ and $\mathcal{NP}$ of problems solvable in polynomial time by deterministic and non-deterministic Turing machines, respectively.

Problems Solvable in Polynomial Time

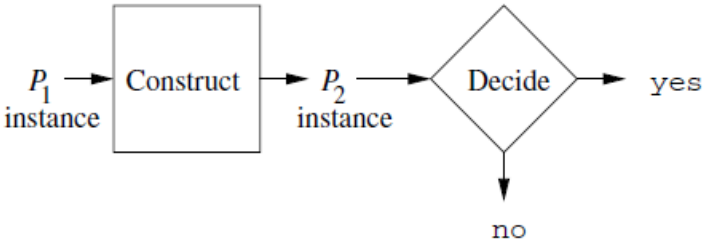
- A Turing machine M is said to be of time complexity $T(n)$ if whenever M is given an input w of length n, M halts after making at most $T(n)$ moves, regardless of whether or not M accepts.
- We say a language L is in class $\mathcal{P}$ if there is some polynomial $T(n)$ such that $L = L(M)$ for some deterministic TM M of time complexity $T(n)$.
- Many problems have efficient solutions. These problems are generally in $\mathcal{P}$. One such problem is : finding a minimum-weight spanning tree for a graph.

Nondeterministic Polynomial Time

- A fundamental class of problems that can be solved by a nondeterministic TM that runs in polynomial time.
- Formally, we say a language L is in the class $\mathcal{NP}$ (nondeterministic polynomial) if there is a nondeterministic TM M and a polynomial time complexity $T(n)$ such that $L = L(M)$, and when M is given an input of length n, there are no sequences of more than $T(n)$ moves of M.
- Since every deterministic TM is a nondeterministic TM that happens never to have a choice of moves, $\mathcal{P} \subseteq \mathcal{NP}$.
- However, it appears that $\mathcal{NP}$ contains many problems not in $\mathcal{P}$. The intuitive reason is that a NTM running in polynomial time has the ability to guess an exponential number of possible solutions to a problem and check each one in polynomial time.
- It is one of the deepest open questions of Mathematics whether $\mathcal{P} = \mathcal{NP}$, i.e., whether in fact everything that can be done in polynomial time by a NTM can in fact be done by a DTM in polynomial time, perhaps with a higher-degree polynomial.
- An $\mathcal{NP}$ Example : The Traveling Salesman Problem,
  CLIQUE Problem,
  SUBSET-SUM Problem.

Polynomial-Time Reducibility

- The principle methodology for proving that a problem P_2 cannot be solved in polynomial time (i.e., P_2 not in $\mathcal{P}$) is the reduction of a problem P_1, which is known not to be in $\mathcal{P}$, to P_2.



- Suppose we want to prove the statement "if P_2 is in $\mathcal{P}$, then so is P_1". Since we claim that P_1 is not in $\mathcal{P}$, we could then claim that P_2 is not in $\mathcal{P}$ either.

Polynomial-Time Reducibility

- **Definition :** A function $f : \Sigma^* \to \Sigma^*$ is a **polynomial time computable function** if some polynomial time Turing machine M exists that halts with just $f(w)$ on its tape, when started on any input w.

- **Definition :** Language A is **polynomial time mapping reducible**, or simply **polynomial time reducible**, to language B, written $A \leq_P B$, if a polynomial time computable function $f : \Sigma^* \to \Sigma^*$ exits, where for every w,

  $$w \in A \Leftrightarrow f(w) \in B$$

  The function f is called the **polynomial time reduction** of A to B.

- **Theorem :** If $A \leq_P B$ and $B \in \mathcal{P}$, then $A \in \mathcal{P}$.

NP-Complete Problems

- **Definition :** Let L be a language (problem) in $\mathcal{NP}$. We say L is ***NP-Complete*** if the following statements are true about L :
  1. L is in $\mathcal{NP}$.
  2. For every language $L^{'}$ in $\mathcal{NP}$ there is a polynomial-time reduction of $L^{'}$ to L.

- Some problems L are so hard that although we can prove condition (2) of the definition of NP-Completeness (every language in $\mathcal{NP}$ reduces to L in polynomial time), we cannot prove condition (1) : that L is in $\mathcal{NP}$. If so, we call L ***NP-hard***.

- **Theorem :** If B is NP-complete and $B \leq_P C$ for C in $\mathcal{NP}$, then C is NP-complete.

- **Theorem :** If some NP-complete problem P is in $\mathcal{P}$, then $\mathcal{P} = \mathcal{NP}$.

The Satisfiability Problem

- The *boolean expressions* are build from :
  1. Variables whose values are boolean; i.e., they either have the value 1 (true) or 0 (false).
  2. Binary $\wedge$ and $\vee$, standing for the logical AND and OR of two expressions.
  3. Unary operator $\neg$ standing for logical negation.
  4. Parentheses to group operators and operands, if necessary to alter the default precedence of operators : $\neg$, then $\wedge$, and finally $\vee$.

- An example of a boolean expression is $x \wedge \neg(y \vee z)$.
  - The expression is true exactly when x is true, y is false, and z is false.

- A truth assignment T satisfies boolean expression E if $E(T) = 1$; i.e., the truth assignment T makes expression E true. A boolean expression E is said to be **satisfiable** if there exists at least one truth assignment T that satisfies E.

- The **satisfiability problem** is : " Given a boolean expression, is it satisfiable ?"

NP-Completeness of the SAT Problem

- **Theorem :** (**Cook-Levin Theorem**) SAT is NP-complete.

- **Proof Idea :** Showing that SAT is in NP is easy. The hard part of the proof is showing that any language in $\mathcal{NP}$ is polynomial time reducible to SAT.

  To do so, we construct a polynomial time reduction for each language A in $\mathcal{NP}$ to SAT. The reduction for A takes a string w and produces a Boolean formula ϕ that simulates the NP machine for A on input w. If the machine accepts, ϕ has a satisfying assignment that corresponds to the accepting computation. If the machine doesn't accept, no assignment satisfies ϕ. Therefore, w is in A if and only if ϕ is satisfiable.

NP-Completeness of the SAT Problem

- A restricted satisfiability problem is *CSAT problem*.
- A boolean expression is said to be in *conjunction normal form* or CNF, if it is the AND of clauses.
- CSAT is the problem : given a boolean expression in CNF, is it satisfiable ?
- kSAT is the problem : given a boolean expression in k-CNF, is it satisfiable ?
- **Theorem :** CSAT is NP-complete.
- **Theorem :** 3SAT problem is NP-complete.
- CSAT, 3SAT, and kSAT for all k higher than 3 are NP-complete. However, there are linear-time algorithms for 1SAT and 2SAT.

Example of NP-Complete Problems

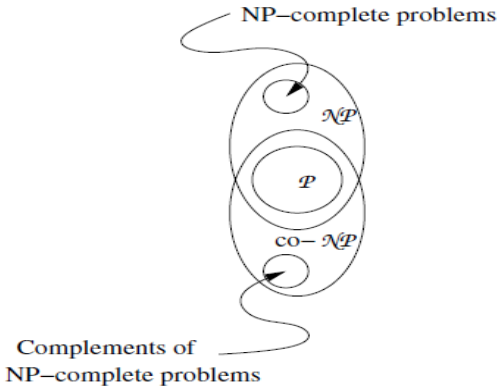
- CLIQUE problem is NP-complete.
- Vertex-Cover Problem is NP-complete.
- The node-cover problem is NP-complete.
- Hamiltonian path problem is NP-complete.
- Subset-Sum problem is NP-complete.

Complements of Languages in $\mathcal{P}$ and $\mathcal{NP}$

- The class of languages $\mathcal{P}$ is closed under complementation.
- *The argument is :* Let L be in $\mathcal{P}$ and let M be a TM for L. Modify M as follows, to accept $\overline{L}$. Introduce a new accepting state q and have the new TM transition to q whenever M halts in a state that is not accepting. Make the former accepting states of M be nonaccepting. Then the modified TM accepts $\overline{L}$, and runs in the same amount of time that M does, with the possible addition of one move. Thus, $\overline{L}$ is in $\mathcal{P}$ if L is.
- It is not known whether $\mathcal{NP}$ is closed under complementation. It appears not, however, and in particular we expect that whenever a language L is NP-complete, then its complement is not in $\mathcal{NP}$.

Complements of Languages in $\mathcal{P}$ and $\mathcal{NP}$

- **Co-$\mathcal{NP}$:** Co-$\mathcal{NP}$ is the set of languages whose complements are in $\mathcal{NP}$.

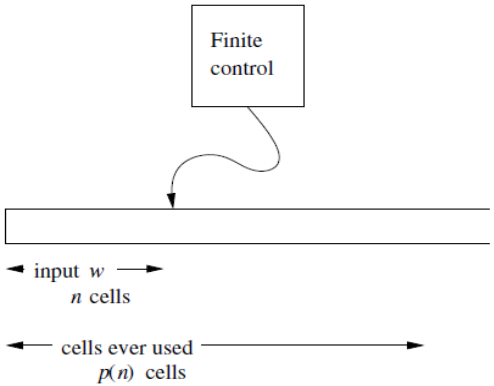


Suspected relationship between co-$\mathcal{NP}$ and other classes of languages

- **Theorem :** $\mathcal{NP}$ = co-$\mathcal{NP}$ if and only if there is some NP-complete problem whose complement is in $\mathcal{NP}$.

Problem Solvable in Polynomial Space

- A polynomial-space-bounded Turing machine is shown in figure.



- There is some polynomial $p(n)$ such that when given input w of length n, the TM never visits more than $p(n)$ cells of its tape. We may assume that the tape is semi-infinite, and the TM never moves left from the beginning of its input.

Problem Solvable in Polynomial Space

- **Class $\mathcal{PS}$:** The languages that are $L(M)$ for some polynomial-space-bounded, deterministic Turing machine M.
- **Class $\mathcal{NPS}$:** The languages that are $L(M)$ for some nondeterministic, polynomial-space-bounded Turing machine M.
- $\mathcal{PS} \subseteq \mathcal{NPS}$, since every deterministic Turing machine is technically nondeterministic also.
- An essential property of polynomial-space-bounded TM's is that they can make only an exponential number of moves before they repeat an ID.

Problem Solvable in Polynomial Space

- **Theorem :** If M is a polynomial-space-bounded TM (deterministic or non-deterministic), and $p(n)$ is its polynomial space bound, then there is a constant c such that if M accepts its input w of length n, it does so within $c^{1+p(n)}$ moves.

- **Theorem :** If L is a language in $\mathcal{PS}$ (respectively $\mathcal{NPS}$), then L is accepted by a polynomial-space-bounded deterministic (respectively non-deterministic) TM that halts after making at most $c^{q(n)}$ moves, for some polynomial $q(n)$ and constant $c > 1$.

- **Theorem :** (Savitch's Theorem) $\mathcal{PS} = \mathcal{NPS}$

$\mathcal{PS}$-Completeness

- A problem P is *complete* for $\mathcal{PS}$ ($\mathcal{PS}$**-complete**) if :
  1. P is in $\mathcal{PS}$.
  2. All languages L in $\mathcal{PS}$ are polynomial-time reducible to P.

- **Theorem :** Suppose P is a $\mathcal{PS}$-complete problem. Then :
  1. If P is in $\mathcal{P}$, then $\mathcal{P} = \mathcal{PS}$.
  2. If P is in $\mathcal{NP}$, then $\mathcal{NP} = \mathcal{PS}$

- *Quantified Boolean Formulas :* Example : $(\exists x)((\exists y)(x + \overline{y}) + (\forall z)(\overline{xz}))$

- **Theorem :** QBF is in $\mathcal{PS}$.

- **Theorem :** The problem QBF is $\mathcal{PS}$-complete.

Conjectured relationships among P, NP, PSPACE, and EXPTIME