Standard Turing Machine and its Variations

A standard Turing machine is a mathematical model of computation that manipulates symbols on an infinite tape according to a set of rules[1]. There are several variations of the standard Turing machine, such as:

- A Turing machine with semi-infinite tape, which has a tape that is unbounded only in one direction.
- A Turing machine with stay-option, which can choose to not move the tape head after each transition.
- A Turing machine with multiple tracks, which has a tape that is divided into several tracks and can read and write all of them at once.
- A Turing machine with off-line mode, which has a separate input and output tape that cannot be modified.
- A Turing machine with multitape, which has several tapes and several tape heads that can operate independently.
- A Turing machine with multidimensional tape, which has a tape that is arranged in a grid or a cube and can move in different directions.
- A nondeterministic Turing machine, which can have multiple possible transitions for each state and symbol and can explore them in parallel.

All these variations are equivalent in computational power to the standard Turing machine, meaning that they can simulate each other and recognize the same class of languages.

Variation of Turing Machine

1. Multiple track Turing Machine:

- A k-track Turing machine(for some k>0) has k-tracks and one R/W head that reads and writes all of them one by one.
- A k-track Turing Machine can be simulated by a single track Turing machine

2. Two-way infinite Tape Turing Machine:

- Infinite tape of two-way infinite tape Turing machine is unbounded in both directions left and right.
- Two-way infinite tape Turing machine can be simulated by one-way infinite Turing machine(standard Turing machine).

3. Multi-tape Turing Machine:

- It has multiple tapes and is controlled by a single head.
- The Multi-tape Turing machine is different from k-track Turing machine but expressive power is the same.
- Multi-tape Turing machine can be simulated by single-tape Turing machine.

4. Multi-tape Multi-head Turing Machine:

- The multi-tape Turing machine has multiple tapes and multiple heads
- Each tape is controlled by a separate head
- Multi-Tape Multi-head Turing machine can be simulated by a standard Turing machine.

5. Multi-dimensional Tape Turing Machine:

- It has multi-dimensional tape where the head can move in any direction that is left, right, up or down.
- Multi dimensional tape Turing machine can be simulated by one-dimensional Turing machine

6. Multi-head Turing Machine:

- A multi-head Turing machine contains two or more heads to read the symbols on the same tape.
- In one step all the heads sense the scanned symbols and move or write independently.
- Multi-head Turing machine can be simulated by a single head Turing machine.

7. Non-deterministic Turing Machine:

- A non-deterministic Turing machine has a single, one-way infinite tape.
- For a given state and input symbol has at least one choice to move (finite number of choices for the next move), each choice has several choices of the path that it might follow for a given input string.
- A non-deterministic Turing machine is equivalent to the deterministic Turing machine.

Universal Turing machine

In computer science, a **universal Turing machine** (**UTM**) is a Turing machine capable of computing any computable sequence,[1] as described by Alan Turing in his seminal paper "On Computable Numbers, with an Application to the Entscheidungsproblem". Common sense might say that a universal machine is impossible, but Turing proves that it is possible.[2] He suggested that we may compare a man in the process of computing a real number to a machine which is only capable of a finite number of conditions q 1: q 2 . .... qI; which will be called "m-configurations".[3] He then described the operation of such machine, as described below, and argued:

"It is my contention that these operations include all those which are used in the computation of a number."

Alan Turing introduced the idea of such a machine in 1936–1937. This principle is considered to be the origin of the idea of a stored-program computer used by John von Neumann in 1946 for the "Electronic Computing Instrument" that now bears von Neumann's name: the von Neumann architecture.

Church's Thesis for Turing Machine

Church Turing Thesis:
Turing machine is defined as an abstract representation of a computing device such as hardware in computers. Alan Turing proposed Logical Computing Machines (LCMs), i.e. Turing's expressions for Turing Machines. This was done to define algorithms properly. So, Church made a mechanical method named as 'M' for manipulation of strings by using logic and mathematics. This method M must pass the following statements:

- Number of instructions in M must be finite.
- Output should be produced after performing finite number of steps.
- It should not be imaginary, i.e. can be made in real life.
- It should not require any complex understanding.

Using these statements Church proposed a hypothesis called
Church's Turing thesis
that can be stated as: "The assumption that the intuitive notion of computable functions can be identified with partial recursive functions."
Or in simple words we can say that "Every computation that can be carried out in the real world can be effectively performed by a Turing Machine."
In 1930, this statement was first formulated by Alonzo Church and is usually referred to as Church's thesis, or the Church-Turing thesis. However, this hypothesis cannot be proved. The recursive functions can be computable after taking following assumptions:

1. Each and every function must be computable.
2. Let 'F' be the computable function and after performing some elementary operations to 'F', it will transform a new function 'G' then this function 'G' automatically becomes the computable function.
3. If any functions that follow above two assumptions must be states as computable function.

Recursive and Recursively Enumerable Languages

A **recursive language** is a language that can be decided by a Turing machine, which means that for any input, the machine will always halt and accept or reject the input. A **recursively enumerable language** is a language that can be recognized by a Turing machine, which means that for any input that belongs to the language, the machine will halt and accept it, but for any input that does not belong to the language, the machine may either halt and reject it or loop forever[12]. Recursive languages are a subset of recursively enumerable languages[34].
Some examples of recursive languages are regular languages, context-free languages, and context-sensitive languages. Some examples of recursively enumerable languages that are not recursive are the halting problem, the Post correspondence problem, and the word problem for groups[14].

Recursive Enumerable (RE) or Type -0 Language

RE languages or type-0 languages are generated by type-0 grammars. An RE language can be accepted or recognized by Turing machine which means it will enter into final state for the strings of language and may or may not enter into rejecting state for the strings which are not part of the language. It means TM can loop forever for the strings which are not a part of the language. RE languages are also called as Turing recognizable languages.

Recursive Language (REC)

A recursive language (subset of RE) can be decided by Turing machine which means it will enter into final state for the strings of language and rejecting state for the strings which are not part of the language. e.g.; L= $\{a^nb^nc^n | n>=1\}$ is recursive because we can construct a turing machine which will move to final state if the string is of the form $a^nb^nc^n$ else move to non-final state. So the TM will always halt in this case. REC languages are also called as Turing decidable languages. The relationship between RE and REC languages can be shown in Figure 1.

-

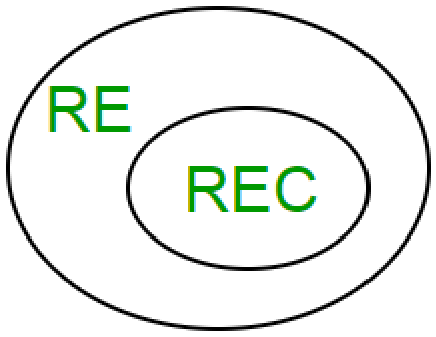


Figure 1

As noted above, every context-sensitive language is recursive. Thus, a simple example of a recursive language is the set *L={abc, aabbcc, aaabbbccc, ...}*; more formally, the set

$$L=\{\omega€\{abc\}^*|\ \omega=a^n b^n c^n \text{ forsome } n\geq1\}$$

is context-sensitive and therefore recursive.

Examples of decidable languages that are not context-sensitive are more difficult to describe. For one such example, some familiarity with mathematical logic is required: Presburger arithmetic is the first-order theory of the natural numbers with addition (but without multiplication). While the set of well-formed formulas in Presburger arithmetic is context-free, every deterministic Turing machine accepting the set of true statements in Presburger

arithmetic has a worst-case runtime of at least , for some constant *c*>0 (Fischer & Rabin 1974). Here, *n* denotes the length of the given formula. Since every context-sensitive language can be accepted by a linear bounded automaton, and such an automaton can be

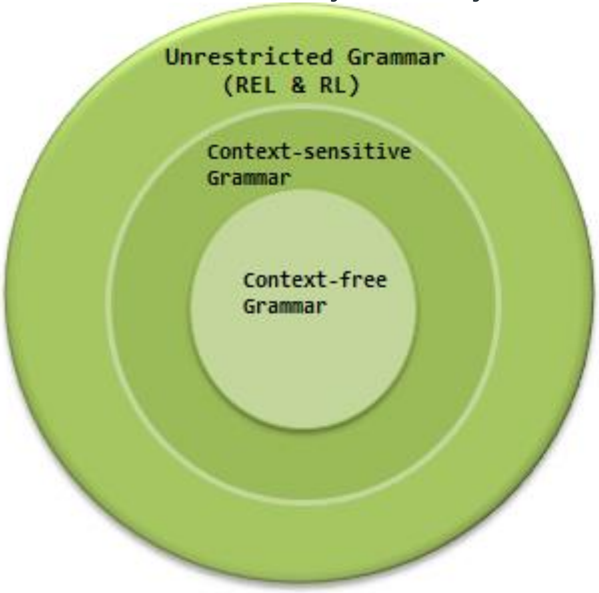
simulated by a deterministic Turing machine with worst-case running time at most for some constant *c*[*citation needed*], the set of valid formulas in Presburger arithmetic is not context-sensitive. On positive side, it is known that there is a deterministic Turing machine running in time at most triply exponential in *n* that decides the set of true formulas in Presburger arithmetic (Oppen 1978). Thus, this is an example of a language that is decidable but not context-sensitive.

Context-Sensitive Grammar –
A Context-sensitive grammar is an Unrestricted grammar in which all the productions are of form –

$$\alpha \rightarrow \beta$$

where $\alpha, \beta \in (V \cup T)^{+}$ and $|\alpha| \leq |\beta|$

Where α and β are strings of non-terminals and terminals.
Context-sensitive grammars are **more powerful** than context-free grammars because there are some languages that can be described by CSG but not by context-free grammars and CSL are less powerful than Unrestricted grammar. That's why context-sensitive grammars are positioned between context-free and unrestricted grammars in the Chomsky hierarchy.



Context-sensitive grammar has 4-tuples. **G = {N, Σ, P, S}**, Where
N = Set of non-terminal symbols
Σ = Set of terminal symbols
S = Start symbol of the production
P = Finite set of productions
All rules in P are of the form $\alpha_1 A \alpha_2 \rightarrow \alpha_1 \beta \alpha_2$

Context-sensitive Language: The language that can be defined by context-sensitive grammar is called CSL. Properties of CSL are :

- Union, intersection and concatenation of two context-sensitive languages is context-sensitive.
- Complement of a context-sensitive language is context-sensitive.

Example –
Consider the following CSG.
S → abc/aAbc
Ab → bA

Ac → Bbcc
bB → Bb
aB → aa/aaA
What is the language generated by this grammar?
Solution:
S → aAbc
→ abAc
→ abBbcc
→ aBbbcc
→ aaAbbcc
→ aabAbcc
→ aabbAcc
→ aabbBbccc
→ aabBbbccc
→ aaBbbbccc
→ aaabbbccc
The language generated by this grammar is {$\mathbf{a^n b^n c^n}$ | n≥1}.

Unrestricted Grammar

In automaton, **Unrestricted Grammar** or **Phrase Structure Grammar** is the most general in the **Chomsky Hierarchy of classification**. This is **type0** grammar, generally used to generate **Recursively Enumerable languages**. It is called unrestricted because no other restriction is made on this except each of their left-hand sides being non-empty. The left-hand sides of the rules can contain terminal and non-terminal, but the condition is at least one of them must be non-terminal. A **Turning Machine** can simulate **Unrestricted Grammar** and **Unrestricted Grammar** can simulate **Turning Machine** configurations. It can always be found for the language recognized or generated by any **Turning Machine**.

Formal Definition

The unrestricted grammar is 4 tuple -

$$G = (N, \Sigma, P, S)$$

N - A finite set of **non-terminal** symbols or **variables**,
Σ - It is a set of terminal symbols or the alphabet of the language being described, where $N \cap \Sigma = \varphi$,
P - It is a finite set of "**productions**" or "**rules**",
S - It is a **start variable** or **non-terminal** symbol.
If, α and β are two strings over the alphabet $N \cup \Sigma$. Then, the rules or productions are of the form $\alpha \rightarrow \beta$. The start variable **S** appears on the left side of the rule.

Example of Unrestricted Grammar

Language

L={$a^n b^n c^n$ | n≥0}

Grammar

S→aBSc {Equal Number of a's, B's, c's}

S→ ε {Eliminate S}

Ba→aB {Move a's to Right of B's}

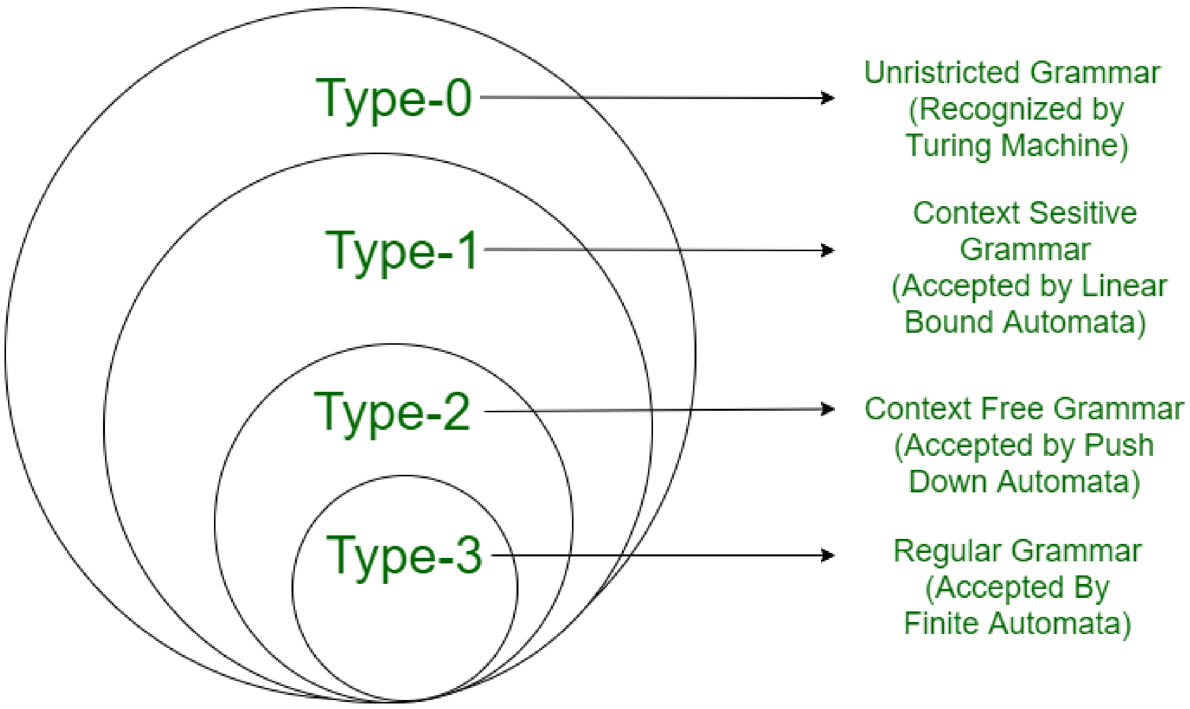
Bc→bc {Reduce B before first c to b}

Bb→bb {Reduce all remaining B's to b}

Chomsky Hierarchy in Theory of Computation

According to Chomsky hierarchy, grammar is divided into 4 types as follows:

1. Type 0 is known as unrestricted grammar.
2. Type 1 is known as context-sensitive grammar.
3. Type 2 is known as a context-free grammar.
4. Type 3 Regular Grammar.


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

Type 0: Unrestricted Grammar:
Type-0 grammars include all formal grammar. Type 0 grammar languages are recognized by turing machine. These languages are also known as the Recursively Enumerable languages.

Grammar Production in the form of   where

```
\alpha          is ( V + T)* V ( V + T)*
V : Variables
T : Terminals.

  is ( V + T )*.
```

In type 0 there must be at least one variable on the Left side of production.

For example:

```
Sab --> ba
A --> S
```

Here, Variables are S, A, and Terminals a, b.

Type 1: Context-Sensitive Grammar
Type-1 grammars generate context-sensitive languages. The language generated by the grammar is recognized by the Linear Bound Automata
In Type 1

- First of all Type 1 grammar should be Type 0.
- Grammar Production in the form of

```
|\alpha |<=|\beta |
```

That is the count of symbol in is less than or equal to

Also β ∈ $(V + T)^+$
i.e. β can not be ε

For Example:

```
S --> AB

AB --> abc

B --> b
```

Type 2: Context-Free Grammar: Type-2 grammars generate context-free languages. The language generated by the grammar is recognized by a Pushdown automata. In Type 2:

- First of all, it should be Type 1.
- The left-hand side of production can have only one variable and there is no restriction on

|\alpha | = 1.

For example:

```
S --> AB

A --> a

B --> b
```

Type 3: Regular Grammar: Type-3 grammars generate regular languages. These languages are exactly all languages that can be accepted by a finite-state automaton. Type 3 is the most restricted form of grammar.
Type 3 should be in the given form only :

```
V --> VT / T          (left-regular grammar)
(or)
V --> TV /T          (right-regular grammar)
```

For example:

```
S --> a
```

The above form is called strictly regular grammar.

There is another form of regular grammar called extended regular grammar. In this form:

```
V --> VT* / T*.         (extended left-regular grammar)
```

```
(or)
```

```
V --> T*V /T*           (extended right-regular grammar)
```

For example :

```
S --> ab.
```

Please write comments if you find anything incorrect, or if you want to share more information about the topic discussed above.

Examples of TM

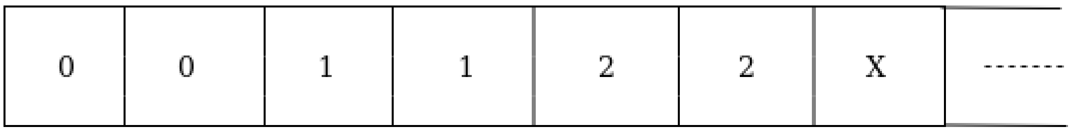
Example 1:

Construct a TM for the language L = {$0^n1^n2^n$} where n≥1

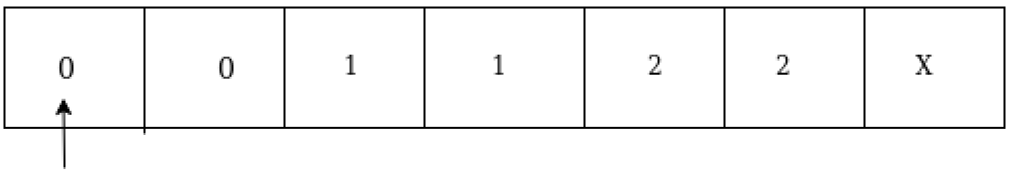
Solution:

L = {$0^n1^n2^n$ | n≥1} represents language where we use only 3 character, i.e., 0, 1 and 2. In this, some number of 0's followed by an equal number of 1's and then followed by an equal number of 2's. Any type of string which falls in this category will be accepted by this language.

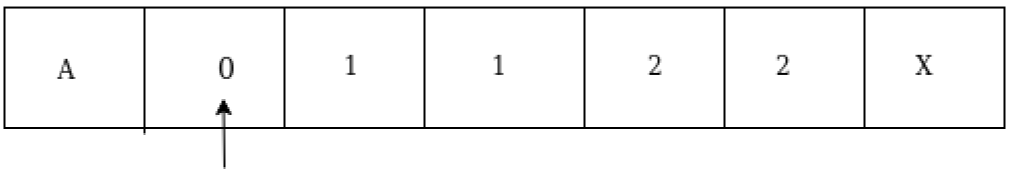
The simulation for 001122 can be shown as below:



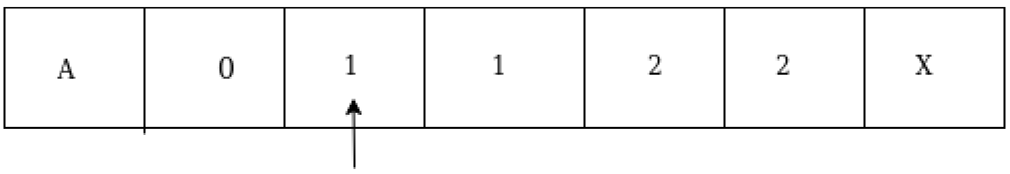
Now, we will see how this Turing machine will work for 001122. Initially, state is q0 and head points to 0 as:



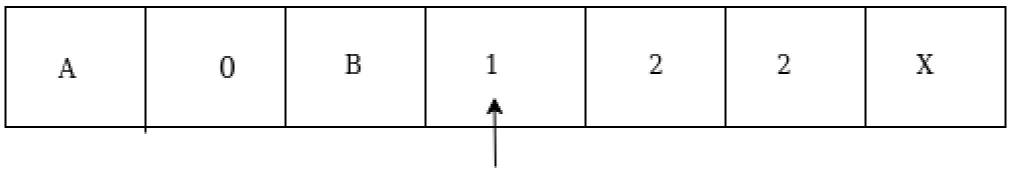
The move will be δ(q0, 0) = δ(q1, A, R) which means it will go to state q1, replaced 0 by A and head will move to the right as:



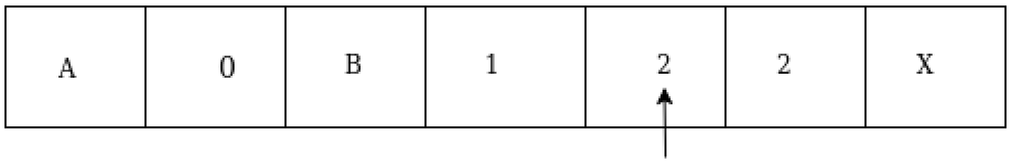
The move will be δ(q1, 0) = δ(q1, 0, R) which means it will not change any symbol, remain in the same state and move to the right as:



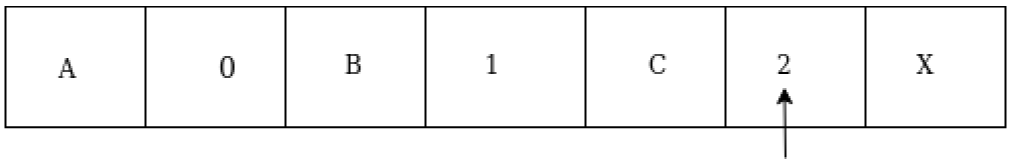
The move will be δ(q1, 1) = δ(q2, B, R) which means it will go to state q2, replaced 1 by B and head will move to right as:



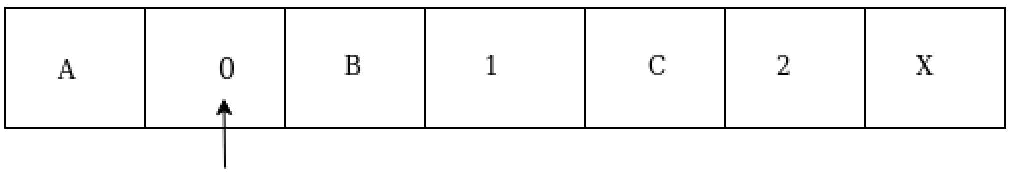
The move will be δ(q2, 1) = δ(q2, 1, R) which means it will not change any symbol, remain in the same state and move to right as:



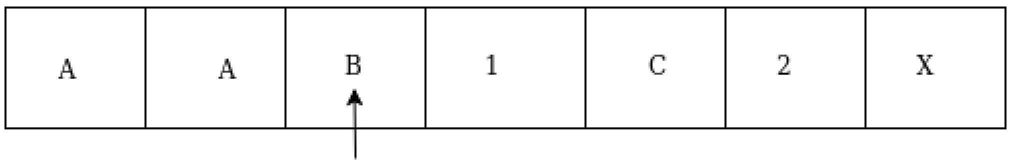
The move will be δ(q2, 2) = δ(q3, C, R) which means it will go to state q3, replaced 2 by C and head will move to right as:



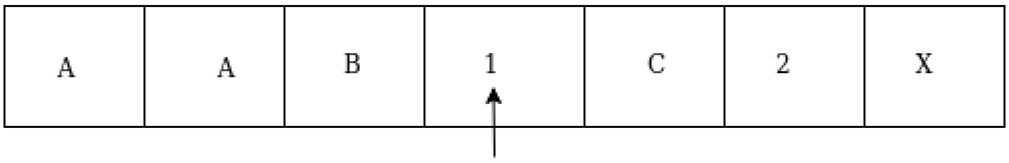
Now move δ(q3, 2) = δ(q3, 2, L) and δ(q3, C) = δ(q3, C, L) and δ(q3, 1) = δ(q3, 1, L) and δ(q3, B) = δ(q3, B, L) and δ(q3, 0) = δ(q3, 0, L), and then move δ(q3, A) = δ(q0, A, R), it means will go to state q0, replaced A by A and head will move to right as:



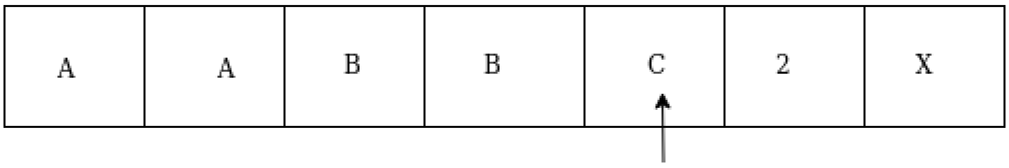
The move will be δ(q0, 0) = δ(q1, A, R) which means it will go to state q1, replaced 0 by A, and head will move to right as:



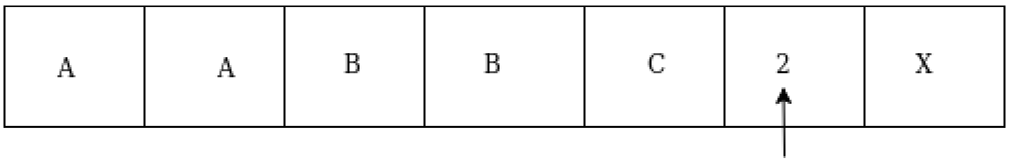
The move will be δ(q1, B) = δ(q1, B, R) which means it will not change any symbol, remain in the same state and move to right as:



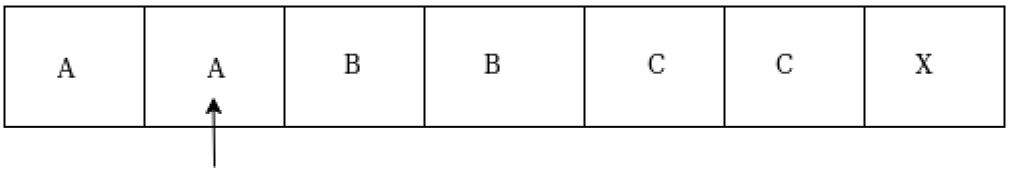
The move will be δ(q1, 1) = δ(q2, B, R) which means it will go to state q2, replaced 1 by B and head will move to right as:



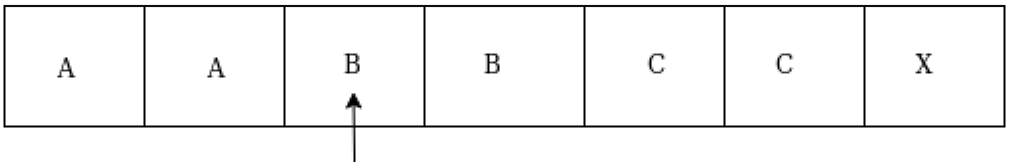
The move will be δ(q2, C) = δ(q2, C, R) which means it will not change any symbol, remain in the same state and move to right as:



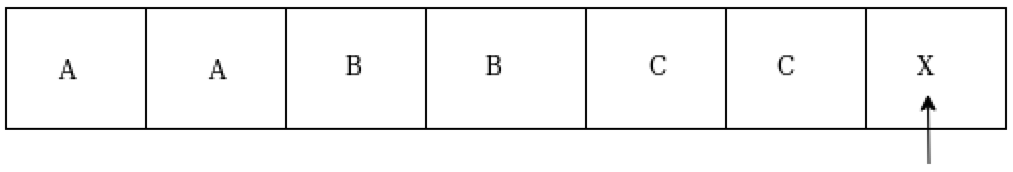
The move will be δ(q2, 2) = δ(q3, C, L) which means it will go to state q3, replaced 2 by C and head will move to left until we reached A as:



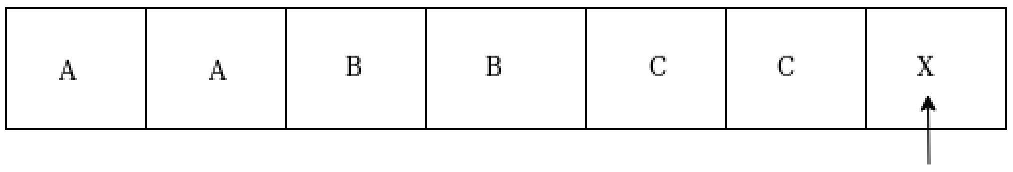
immediately before B is A that means all the 0's are market by A. So we will move right to ensure that no 1 or 2 is present. The move will be δ(q2, B) = (q4, B, R) which means it will go to state q4, will not change any symbol, and move to right as:



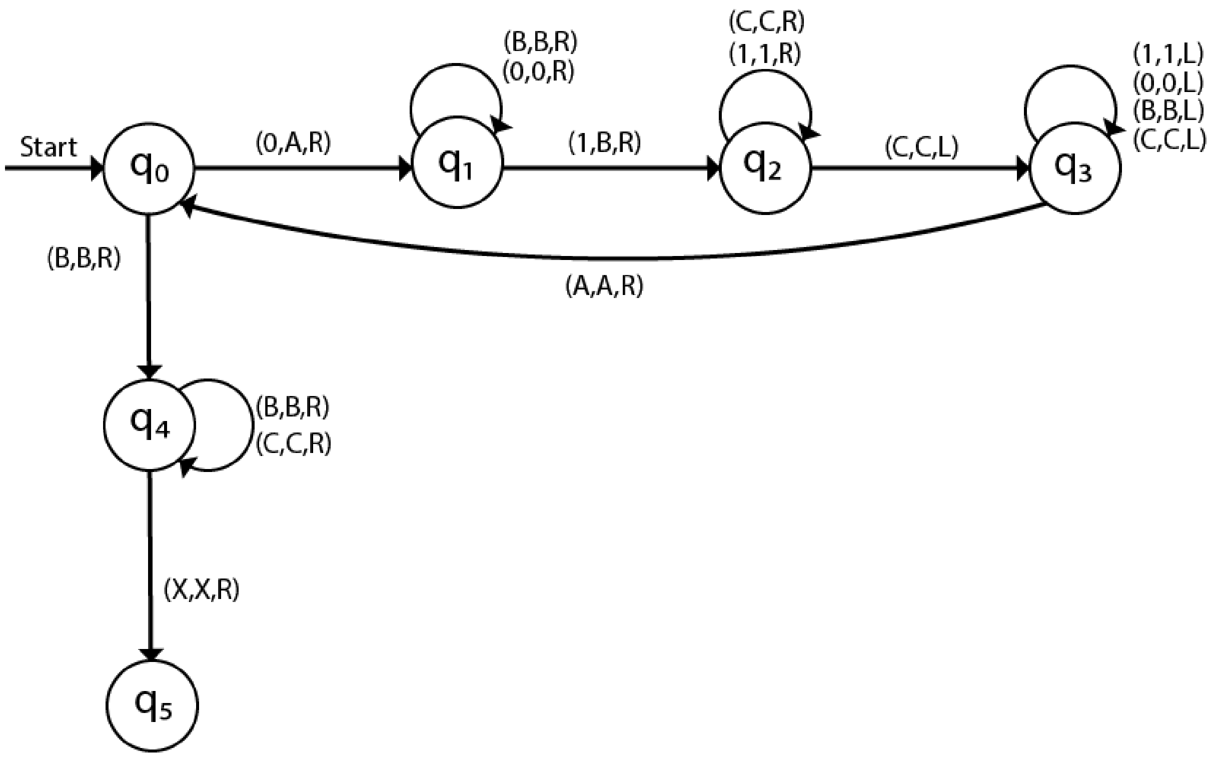
The move will be (q4, B) = δ(q4, B, R) and (q4, C) = δ(q4, C, R) which means it will not change any symbol, remain in the same state and move to right as:



The move δ(q4, X) = (q5, X, R) which means it will go to state q5 which is the HALT state and HALT state is always an accept state for any TM.



The same TM can be represented by Transition Diagram:


Start
q0
(0,A,R)
q1
(B,B,R)
(0,0,R)
(1,B,R)
q2
(C,C,R)
(1,1,R)
(C,C,L)
q3
(1,1,L)
(0,0,L)
(B,B,L)
(C,C,L)
(A,A,R)
(B,B,R)
q4
(B,B,R)
(C,C,R)
(X,X,R)
q5