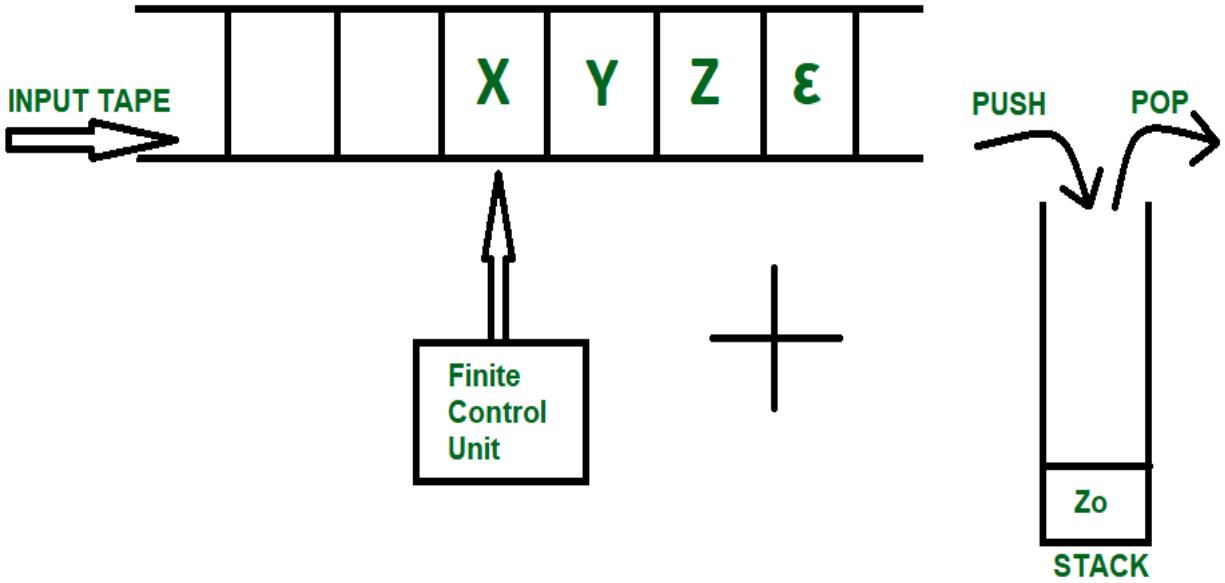
Push Down Automata

Pushdown Automata is a finite automata with extra memory called stack which helps Pushdown automata to recognize Context Free Languages.

A Pushdown Automata (PDA) can be defined as :

- Q is the set of states
- ∑is the set of input symbols
- Γ is the set of pushdown symbols (which can be pushed and popped from stack)
- q0 is the initial state
- Z is the initial pushdown symbol (which is initially present in stack)
- F is the set of final states
- δ is a transition function which maps Q x {Σ ∪ ∈} x Γ into Q x Γ*. In a given state, PDA will read input symbol and stack symbol (top of the stack) and move to a new state and change the symbol of stack.



The diagram above shows a input tape which is how a Finite Automata works, the strings are accepted into the tape and the read header keeps getting updated according the instructions provided by Finite Control Unit. Pushdown Automata on the other hand is a combination of this tape and a Stack data structure.

This assumption helps us in two ways :-

1. We overcome the underflow condition thus saving any memory to keep a check on Stack empty.
2. Initial Stack symbol can be used to declare that string processing has been done successfully.

Deterministic PDA :-

$\delta : Q \times \sum \times \Gamma = Q \times \Gamma^*$

Non-Deterministic PDA :-

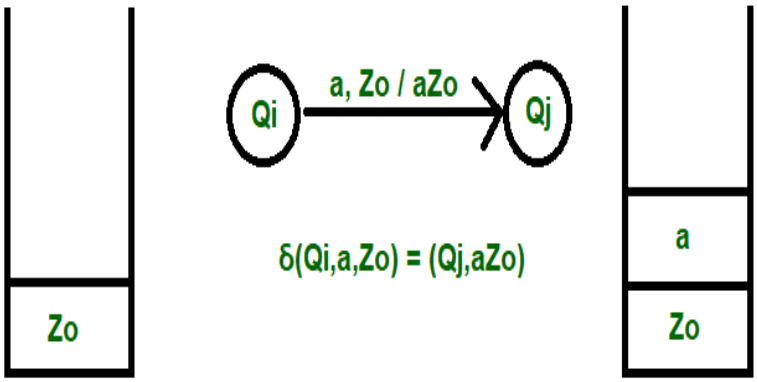
$\delta : Q \times \sum \times \Gamma = 2^{(Q \times \Gamma^*)}$

Tau Symbol (Γ) is used to denote all the Stack Alphabets. Each input alphabet (same or different) can be denoted by a different Stack symbol. It's also necessary as it conveys the topmost element of the stack to the machine.

Delta Function (δ)) is the transition function, the use of which will become more clear by taking a closer look at the Three Major operations done on Stack :-

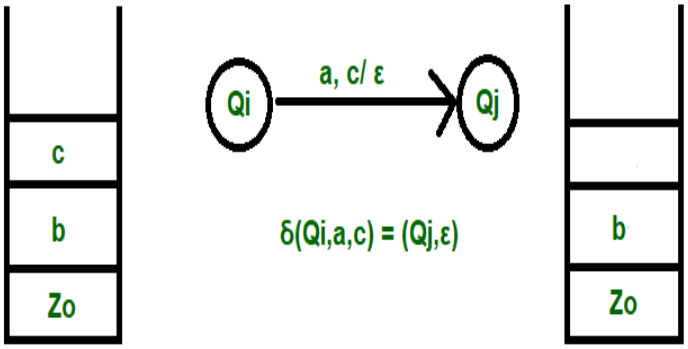
1. Push
2. Pop
3. Skip

1.PUSH



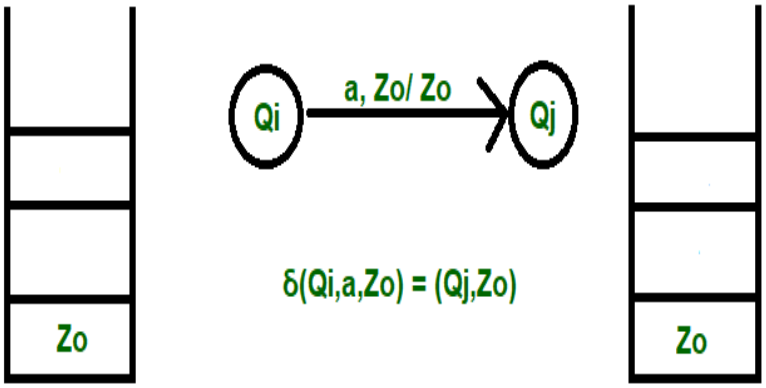
Push Operation is done as shown in the diagram.
The transition takes place in the order :- *Input, Topmost Element / Final List*
Here, a is the input element, which is inserted into the stack, thus making the final content to be aZo.

2.POP



Pop Operation is done as shown in the diagram.

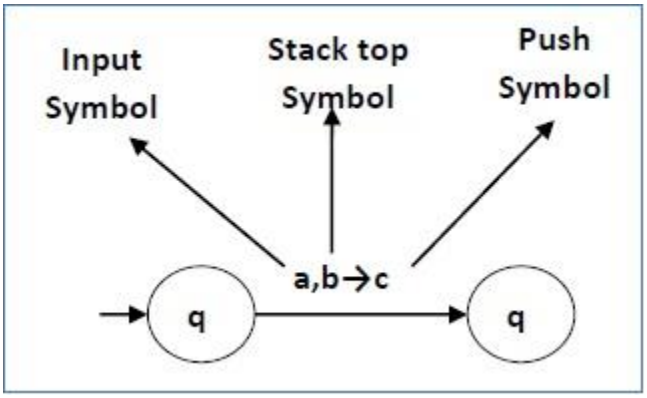
The transition takes place in the order :- *Input Element, Topmost Element / Removal Confirmation*
Here, a is the input, c is the element to be deleted and the removal confirmation is shown by Epsilon symbol declaring that the immediate has been popped and is Empty.
3.SKIP



Skip operation is done as shown in the diagram.

The transition takes place in the order:- *Input Element, Topmost element/Topmost Element*
Here, a is the input and the stack remains unchanged after this operation.

Hence, this concludes the detailed study on how a Pushdown Automata works. Next we will be heading onto some examples to make working and transitions more clear



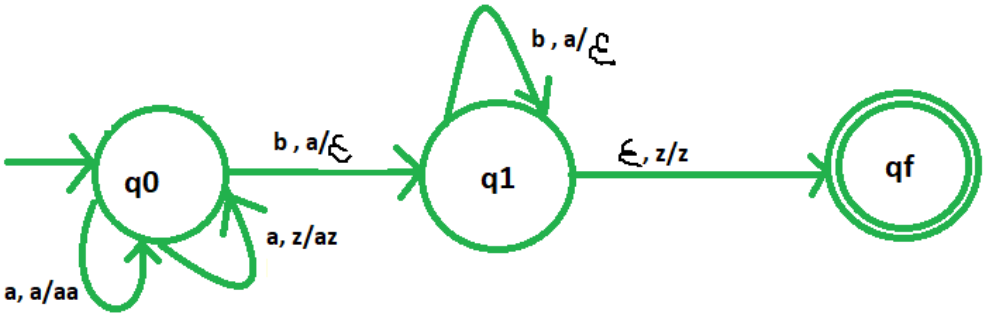
This means at state q_1, if we encounter an input string **'a'** and top symbol of the stack is **'b'**, then we pop **'b'**, push **'c'** on top of the stack and move to state q_2.

Turnstile notation
⊢ sign is called a "turnstile notation" and represents
one move.
⊢* sign represents a sequence of moves.
Eg- (p, b, T) ⊢ (q, w, α)
This implies that while taking a transition from state p to state q, the input symbol 'b' is consumed, and the top of the stack 'T' is replaced by a new string 'α'

Example : Define the pushdown automata for language $\{a^n b^n \mid n > 0\}$
Solution : M = where Q = { q0, q1 } and Σ = { a, b } and Γ = { A, Z } and &delta is given by :

δ (q0, a, Z) = { (q0, AZ) }
δ (q0, a, A) = { (q0, AA) }
δ (q0, b, A) = { (q1, ∈) }
δ (q1, b, A) = { (q1, ∈) }
δ (q1, ∈, Z) = { (q2, Z) }
Let us see how this automata works for aaabbb.

Row	State	Input	δ(transition function used)	Stack(Leftmost symbol represents top of stack)	State after move
1	q0	aaabbb		Z	q0
2	q0	**a**aabbb	δ(q0,a,Z)={(q0,AZ)}	AZ	q0
3	q0	a**a**abbb	δ(q0,a,A)={(q0,AA)}	AAZ	q0
4	q0	aa**a**bbb	δ(q0,a,A)={(q0,AA)}	AAAZ	q0
5	q0	aaa**b**bb	δ(q0,b,A)={(q1,∈)}	AAZ	q1
6	q1	aaab**b**b	δ(q1,b,A)= {(q1,∈)}	AZ	q1
7	q1	aaabb**b**	δ(q1,b,A)= {(q1,∈)}	Z	q1
8	q1	∈	δ(q1, ∈,Z)= {(q1,∈)}	∈	q1



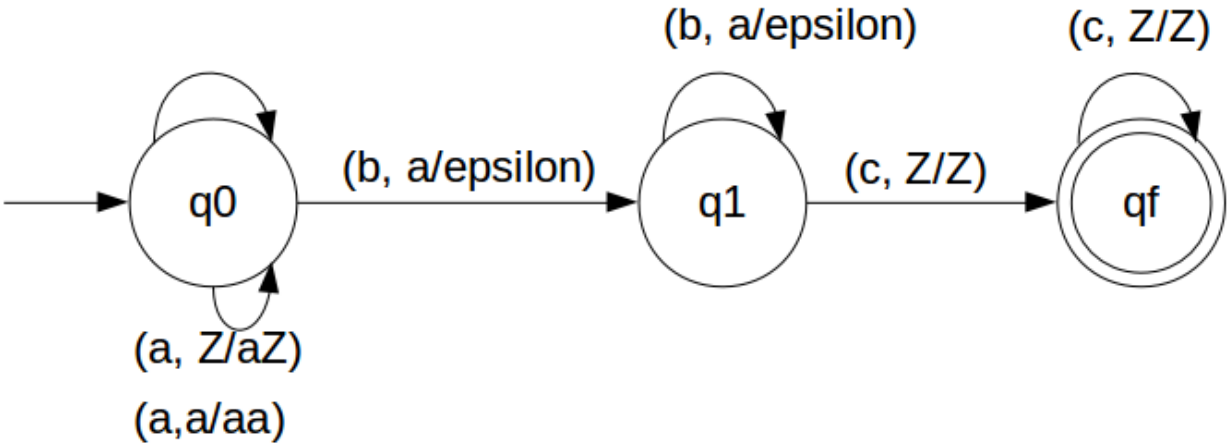
Required PDA

Example 2: L= ($a^n b^n c^m$, where n>=1,m>=0)

First we have to count number of a's and that number should be equal to number of b's. That we will achieve by pushing a's in STACK and then we will pop a's whenever "b" comes.
Then for c we will let happen nothing.

So in the end of the strings if nothing is left in the STACK then we can say that language is accepted in the PDA.

We have designed the PDA for the problem:



STACK Transiton Function

$$\delta(q0, a, Z) = (q0, aZ)$$

$$\delta(q0, a, a) = (q0, aa)$$

$$\delta(q0, b, a) = (q1, \varepsilon)$$

$$\delta(q1, b, a) = (q1, \varepsilon)$$

$$\delta(q1, c, Z) = (qf, Z)$$

$$\delta(qf, c, Z) = (qf, Z)$$

Note: *qf is Final State*

Explanation

Lets see, how this DPDA is working:
We will take one input string: "aabbcc"

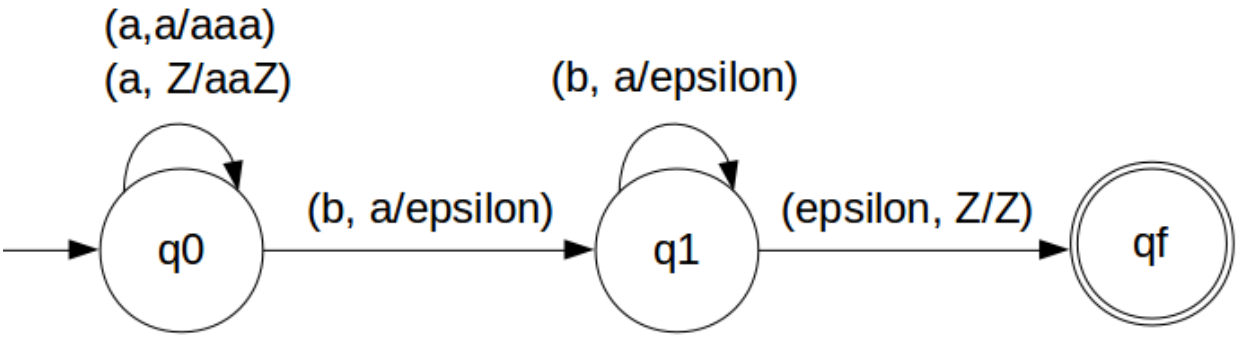
1. Scan string from left to right
2. First input is 'a' and follow the rule:
3. on input 'a' and STACK alphabet Z, push the input 'a' into STACK as : (a,Z/aZ) and state will be q0
4. Second input is 'a' and so follow the rule:
5. on input 'a' and STACK alphabet 'a', push the input 'a' into STACK as : (a,a/aa) and state will be q0
6. Third input is 'b' and so follow the rule:
7. on input 'b' and STACK alphabet 'a', pop STACK with one 'a' as : (b,a/&epsiloon;) and state will be now q1
8. Fourth input is 'b' and so follow the rule:

9. on input 'b' and STACK alphabet 'a' and state is q1, pop STACK with one 'a' as : (b,a/&epsiloon;) and state will be q1
10. Fifth input is 'c' and top of STACK is Z so follow the rule:
11. on input 'c' and STACK alphabet 'Z' and state is q1, do nothing: (c,Z/Z) and state will be qf
12. Sixth input is 'c' and top of STACK is Z so follow the rule:
13. on input 'c' and STACK alphabet 'Z' and state is qf, do nothing: (c,Z/Z) and state will be qf
14. We reached end of the string, so follow the rule:
15. If we are standing at final state, qf then string is accepted in the PDA

Example 3 Design a PDA for accepting a language $\{a^n b^{2n} \mid n>=1\}$.

For every two a's push two a's into STACK cause there are two b's for one 'a'
So by pushing two 'a' we can have 'a' for every 'b'.
That we will achieve by pushing two a's and poping a's for every b
And then pushing c's and poping c's for every d's
So in the end of the strings if nothing is left in the STACK then we can say that language is accepted in the PDA.

We have designed the PDA for the problem:



STACK Transiton Function

$\delta(q0, a, Z) = (q0, aaZ)$

$\delta(q0, a, a) = (q0, aaa)$

$\delta(q0, b, a) = (q1, \varepsilon)$

$\delta(q1, b, a) = (q1, \varepsilon)$

$\delta(q1, \varepsilon, Z) = (qf, Z)$

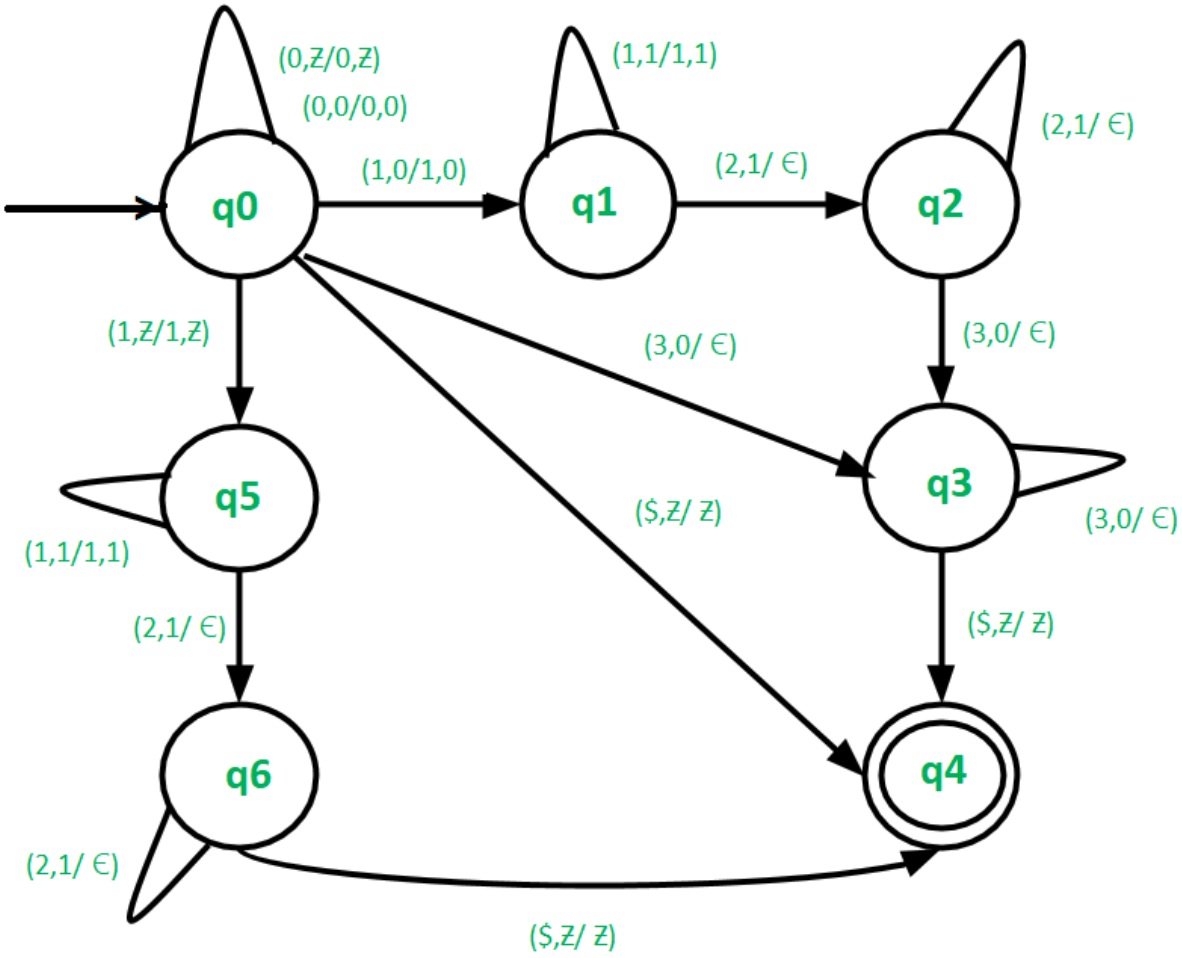
Note: *qf is Final State*

Explanation

Lets see, how this DPDA is working:
We will take one input string: **"aabbbb"**

- Scan string from left to right
- First input is 'a' and follow the rule:
- on input 'a' and STACK alphabet Z, push the two 'a's into STACK as : (a,Z/aaZ) and state will be q0
- Second input is 'a' and so follow the rule:
- on input 'a' and STACK alphabet 'a', push the two 'a's into STACK as : (a,a/aaa) and state will be q0***Now the STACK has "aaaa", So:***
- Third input is 'b' and so follow the rule:
- on input 'b' and STACK alphabet 'a', pop one 'a' from STACK as : (b,a/ε) and state will be q1
- Fourth input is 'b' and so follow the rule:
- on input 'b' and STACK alphabet 'a' and state q1, pop one 'a' from STACK as : (b,a/ε) and state will remain q1
- Fifth input is 'b' and so follow the rule:
- on input 'b' and STACK alphabet 'a', pop one 'a' from STACK as : (b,a/ε) and state will be q1
- Sixth input is 'b' and so follow the rule:
- on input 'b' and STACK alphabet 'a' and state q1, pop one 'a' from STACK as : (b,a/ε) and state will remain q1
- We reached end of the string, so follow the rule:
- on input ε and STACK alphabet Z, go to final state(qf) as : (ε, Z/Z)

Example 4. Construct a PDA for language L = {$0^n1^m2^m3^n$ | n>=1, m>=1}



Context-free languages are **closed** under −

- Union
- Concatenation
- Kleene Star operation

Union

Let L_1 and L_2 be two context free languages. Then $L_1 \cup L_2$ is also context free.

Example

Let $L_1 = \{ a^n b^n , n > 0\}$. Corresponding grammar G_1 will have P: S1 → aAb|ab

Let $L_2 = \{ c^m d^m , m \geq 0\}$. Corresponding grammar G_2 will have P: S2 → cBb| ε

Union of L_1 and L_2, $L = L_1 \cup L_2 = \{ a^n b^n \} \cup \{ c^m d^m \}$

The corresponding grammar G will have the additional production S → S1 | S2

Concatenation

If L_1 and L_2 are context free languages, then L_1L_2 is also context free.

Example

Union of the languages L_1 and L_2, $L = L_1L_2 = \{ a^nb^nc^md^m \}$

The corresponding grammar G will have the additional production $S \rightarrow S1\ S2$

Kleene Star

If L is a context free language, then L* is also context free.

Example

Let $L = \{ a^nb^n , n \geq 0\}$. Corresponding grammar G will have P: $S \rightarrow aAb| \varepsilon$

Kleene Star $L_1 = \{ a^nb^n \}^*$

The corresponding grammar G_1 will have additional productions $S1 \rightarrow SS_1 | \varepsilon$

Context-free languages are **not closed** under −

- **Intersection** − If L1 and L2 are context free languages, then L1 ∩ L2 is not necessarily context free.
- **Intersection with Regular Language** − If L1 is a regular language and L2 is a context free language, then L1 ∩ L2 is a context free language.
- **Complement** − If L1 is a context free language, then L1' may not be context free.

Decision Properties

As usual, when we talk about "a CFL" we really mean "a representation for the CFL, e.g., a CFG or a PDA accepting by final state or empty stack.

There are algorithms to decide if:

1. String w is in CFL L.

2. CFL L is empty.

3. CFL L is infinite.

Turing Machine in TOC

Turing Machine was invented by Alan Turing in 1936 and it is used to accept Recursive Enumerable Languages (generated by Type-0 Grammar).

A turing machine consists of a tape of infinite length on which read and writes operation can be performed. The tape consists of infinite cells on which each cell either contains input symbol or
a special symbol called blank. It also consists of a head pointer which points to cell currently being read and it can move in both directions. A TM is expressed as a 7-tuple (Q, T, B, ∑, δ, q0, F) where:

- **Q** is a finite set of states

- **T** is the tape alphabet (symbols which can be written on Tape)
- **B** is blank symbol (every cell is filled with B except input alphabet initially)
- ∑ is the input alphabet (symbols which are part of input alphabet)
- **δ** is a transition function which maps Q × T → Q × T × {L,R}. Depending on its present state and present tape alphabet (pointed by head pointer), it will move to new state, change the tape symbol (may or may not) and move head pointer to either left or right.
- **q0** is the initial state
- **F** is the set of final states. If any state of F is reached, input string is accepted.

Let us construct a turing machine for $L=\{0^n1^n|n>=1\}$

- Q = {q0,q1,q2,q3} where q0 is initial state.
- T = {0,1,X,Y,B} where B represents blank.
- ∑ = {0,1}
- F = {q3}

Transition function δ is given in Table 1 as:

	0	1	X	Y	B
q0	(q1,X,R)			(q3,Y,R)	
q1	(q1,0,R)	(q2,Y,L)		(q1,Y,R)	
q2	(q2,0,L)		(q0,X,R)	(q2,Y,L)	
q3				(q3,Y,R)	**Halt**

Illustration

Let us see how this turing machine works for 0011. Initially head points to 0 which is underlined and state is q0 as:

B	0	0	1	1	B

The move will be δ(q0, 0) = (q1, X, R). It means, it will go to state q1, replace 0 by X and head will move to right as:

B	X	0	1	1	B

The move will be δ(q1, 0) = (q1, 0, R) which means it will remain in same state and without changing any symbol, it will move to right as:

B	X	0	1	1	B

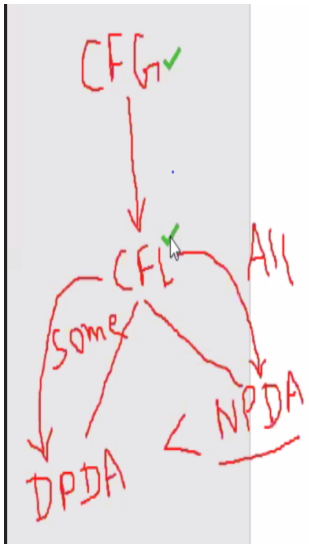
The move will be δ(q1, 1) = (q2, Y, L) which means it will move to q2 state and changing 1 to Y, it will move to left as:

B	X	0	Y	1	B

Working on it in the same way, the machine will reach state q3 and head will point to B as shown:

B	X	X	Y	Y	B

Using move δ(q3, B) = halt, it will stop and accepted.

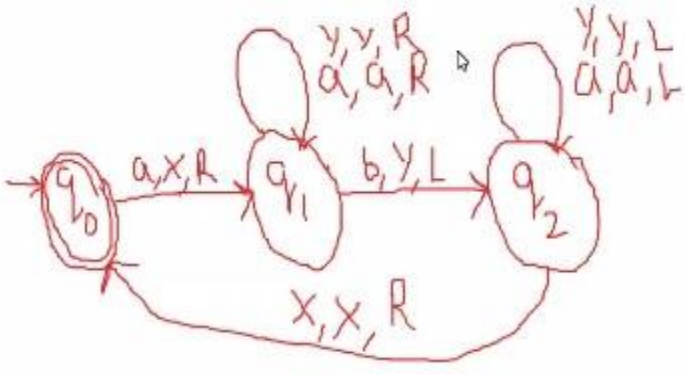


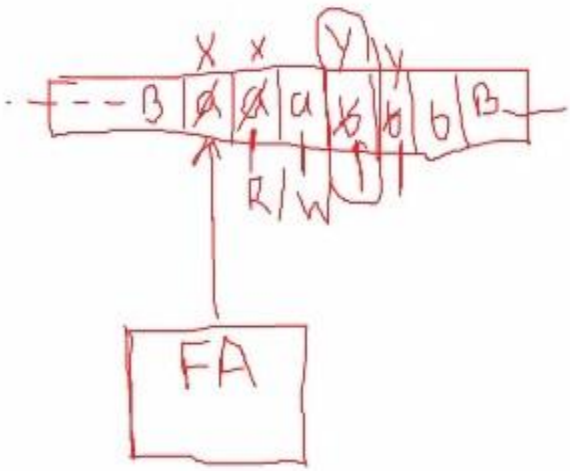
input,top element/Two top symbol → Push
input,top element/E → POP

input,top element(z)/z → Skip

Let us construct a turing machine for L={0n1n|n>=1}

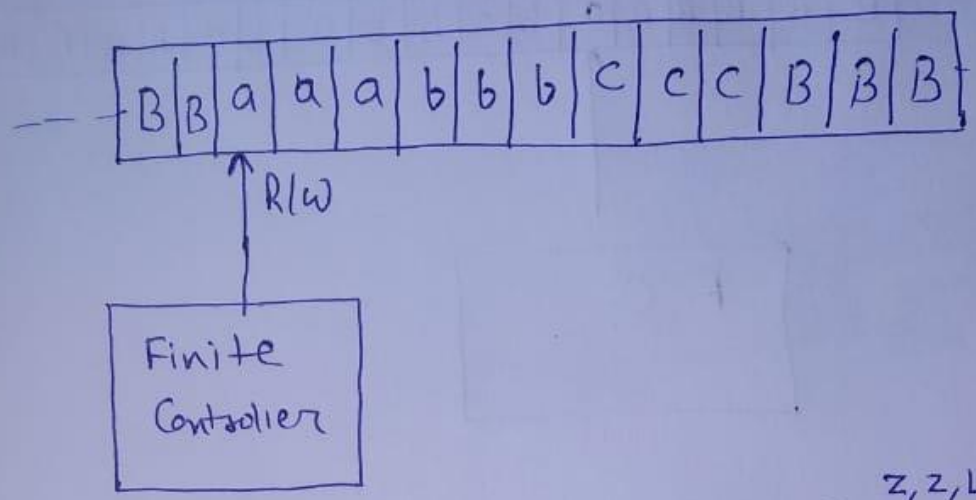
aaabbb

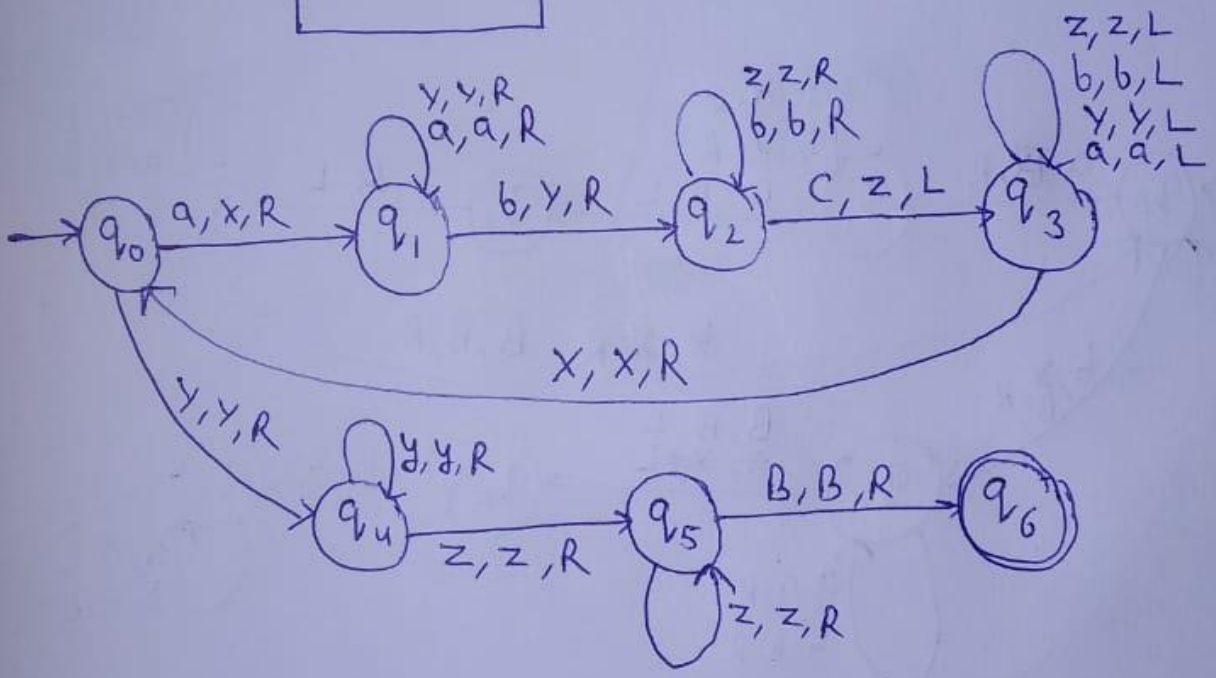




Ex:- $L = \{a^n b^n c^n \mid n \geq 1\}$

String:- aaa bbb ccc

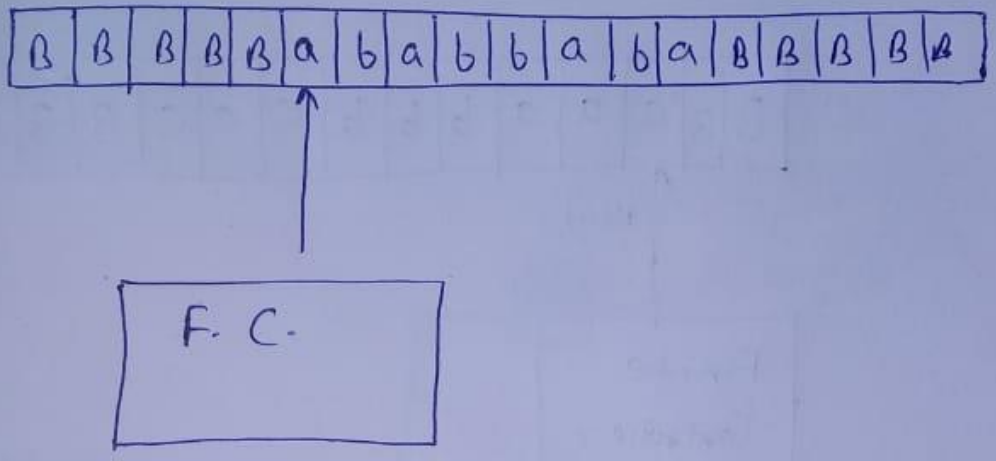


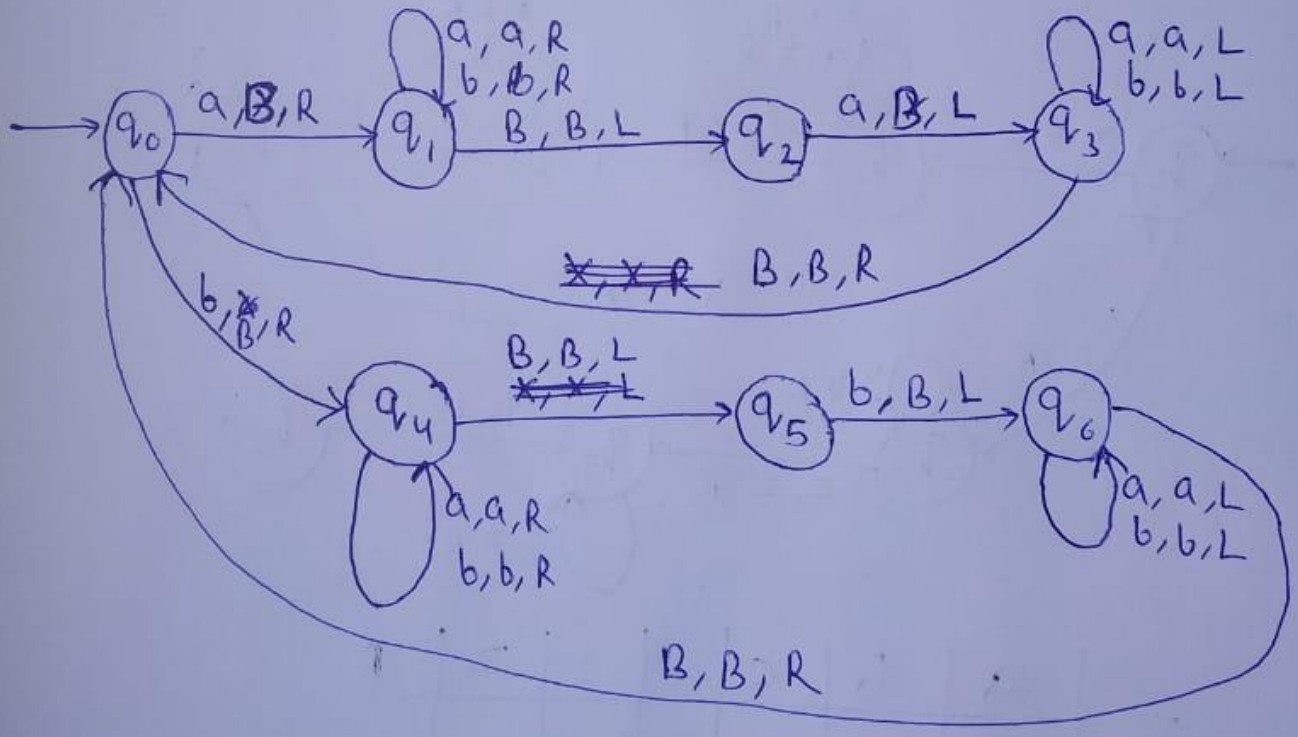


	a	b	c	x	y	z	B
q_0	q_1, X, R				q_4, Y, R		
q_1	q_1, a, R	q_2, Y, R			q_1, y, R		
q_2		q_2, b, R	q_3, z, L			q_2, z, R	
q_3	q_3, a, L	q_3, b, L		q_0, X, R	q_3, Y, L	q_3, z, L	
q_4		~~q_4, b, R~~			q_4, y, R	q_5, z, R	
q_5						q_5, z, R	q_6, B, R
q_6							

Ex:2 Construct the TM for $L=\{ ww^R \mid w \in \{a,b\}\}$

Let assume String :- $\underbrace{abab}_{w}\underbrace{baba}_{w^R}$





Turing Machine for addition

A number is represented in binary format in different finite automatas like 5 is represented as (101) but in case of addition using a turing machine unary format is followed. In unary format a number is represented by either all ones or all zeroes. For example, 5 will be represented by a sequence of five zeroes or five ones. 5 = 1 1 1 1 1 or 0 0 0 0 0. Lets use zeroes for representation.

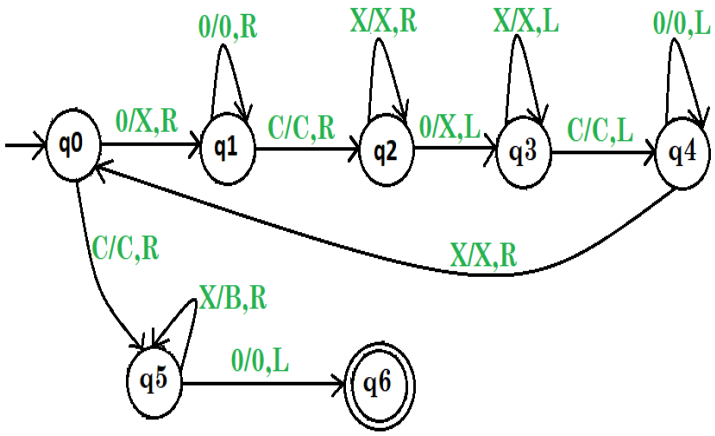
For adding 2 numbers using a Turing machine, both these numbers are given as input to the Turing machine separated by a “c”.

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

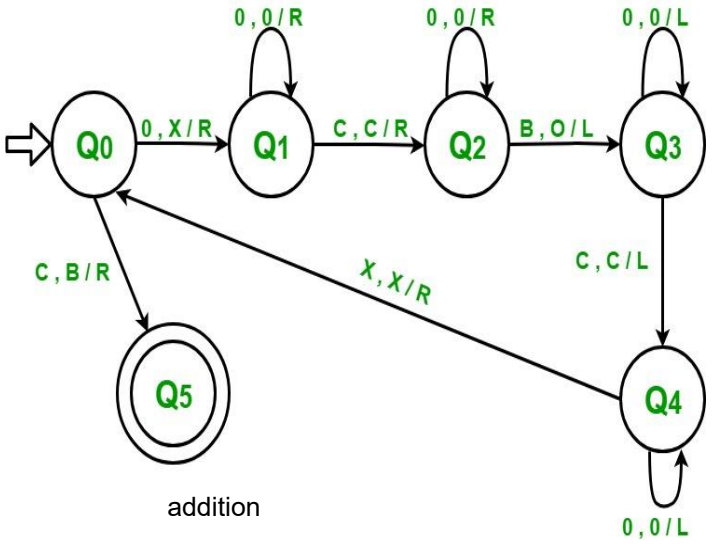
```
Input  :  0 0 c 0 0 0    // 2 + 3
Output :  0 0 0 0 0      // 5


Input  :  0 0 0 0 c 0 0 0  // 4 + 3
Output :  0 0 0 0 0 0 0    // 7
```

Approach used –
Convert a 0 in the first number in to X and then traverse entire input and convert the first blank encountered into 0. Then move towards left ignoring all 0’s and “c”. Come the position just next to X and then repeat the same procedure till the time we get a “c” instead of X on returning. Convert the c into blank and addition is completed.
Steps –
Step-1: Convert 0 into X and goto step 2. If symbol is “c” then convert it into blank(B), move right and goto step-6.
Step-2: Keep ignoring 0’s and move towards right. Ignore “c”, move right and goto step-3.
Step-3: Keep ignoring 0’s and move towards right. Convert a blank(B) into 0, move left and goto step-4.
Step-4: Keep ignoring 0’s and move towards left. Ignore “c”, move left and goto step-3.
Step-5: Keep ignoring 0’s and move towards left. Ignore an X, move left and goto step-1.
Step-6: End.



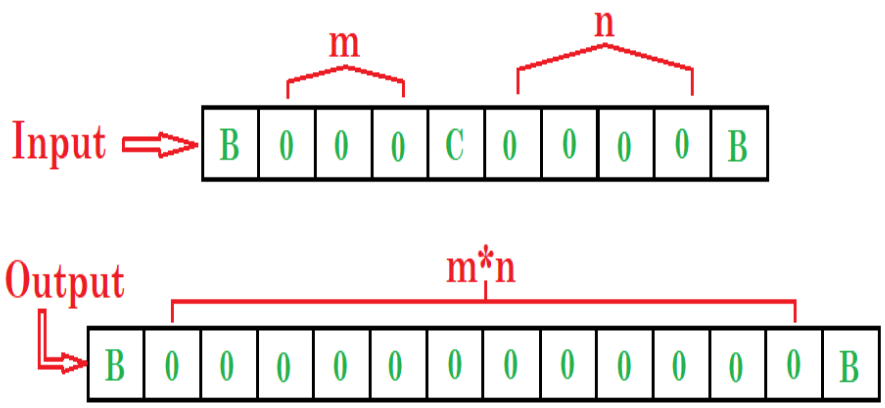
subtraction



addition

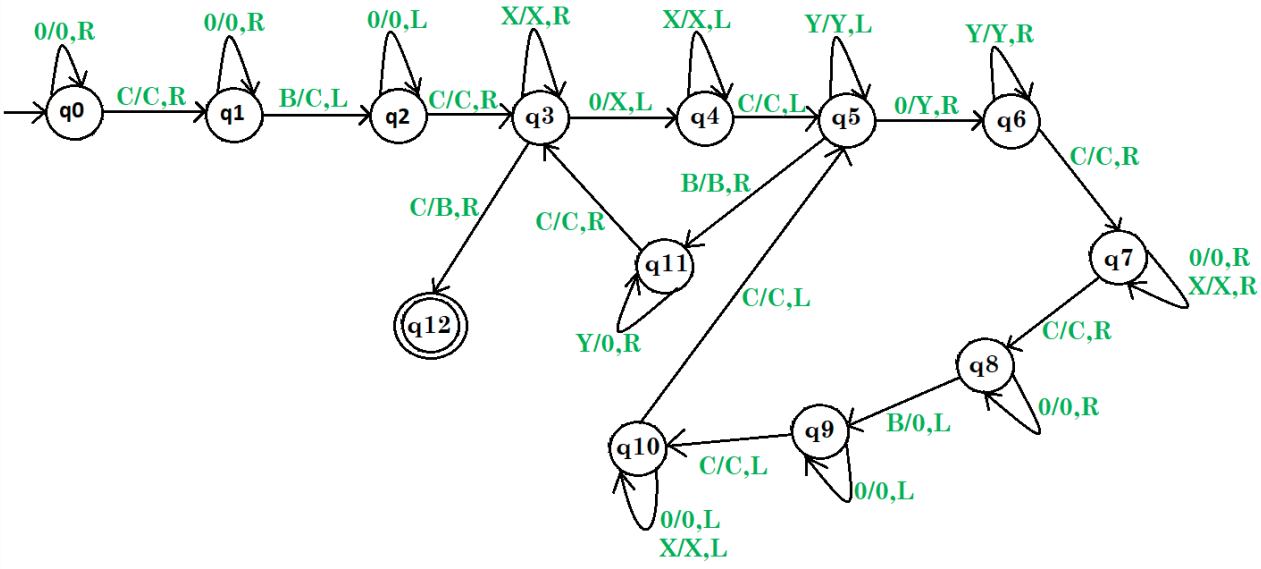
Turing machine for multiplication

Problem: Draw a turing machine which multiply two numbers.
Example:



Steps:

- **Step-1.** First ignore 0's, C and go to right & then if B found convert it into C and go to left.
- **Step-2.** Then ignore 0's and go left & then convert C into C and go right.
- **Step-3.** Then convert all X into X and go right if 0 found convert it into X and go to left otherwise if C found convert it into B and go to right and **stop the machine.**
- **Step-4.** If then X found convert it into X and go left then C into C and left then Y into Y and left.
- **Step-5.** Then if B found convert it into B and right then if Y into 0 and right or if C into C and right and go to step 3 and repeat the process otherwise if 0 found after 4th step then convert it into Y and right then Y into Y and right then C into C and right then 0 into 0 or X into X and right then C into C and right then 0 into 0 and right then B into 0 and left then 0 into 0 and left then C into C and left then 0 into 0 or X into X and left then C into C and left.
- **Step-6.** Then repeat the 5th step.

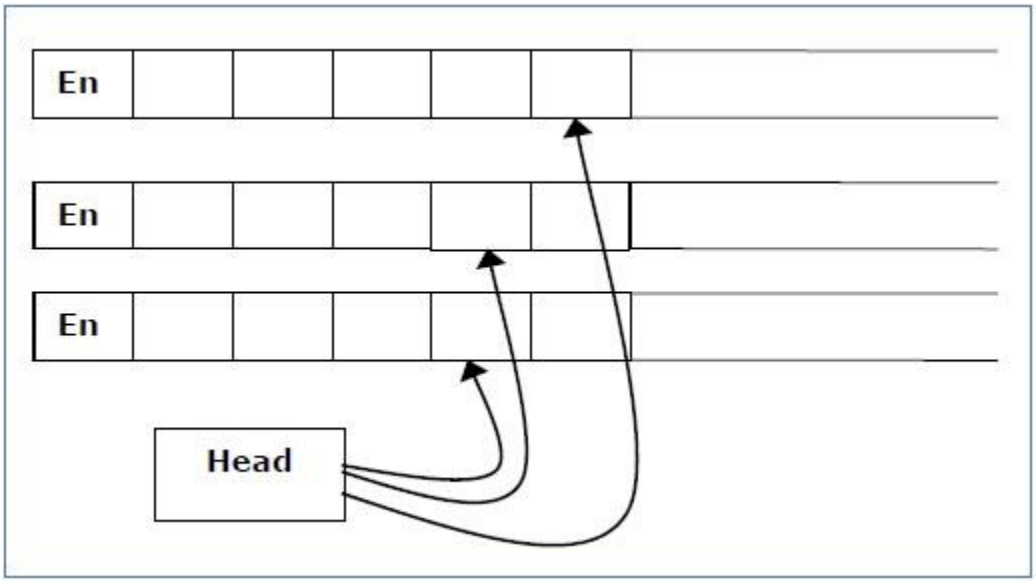


Here, **q0** shows the initial state and **q1, q2, ….., q10, q11**are the transition states and **q12**shows the final state.

And X, Y, 0, C are the variables used for multiplication and R, L shows right and left.

Multi-tape Turing Machine

Multi-tape Turing Machines have multiple tapes where each tape is accessed with a separate head. Each head can move independently of the other heads. Initially the input is on tape 1 and others are blank. At first, the first tape is occupied by the input and the other tapes are kept blank. Next, the machine reads consecutive symbols under its heads and the TM prints a symbol on each tape and moves its heads.



A Multi-tape Turing machine can be formally described as a 6-tuple (Q, X, B, δ, q_0, F) where −

- **Q** is a finite set of states
- **X** is the tape alphabet
- **B** is the blank symbol
- **δ** is a relation on states and symbols where

  $\delta: Q \times X^k \rightarrow Q \times (X \times \{Left\_shift, Right\_shift, No\_shift\})^k$

  where there is **k** number of tapes
- **q_0** is the initial state
- **F** is the set of final states

Note − Every Multi-tape Turing machine has an equivalent single-tape Turing machine.

Universal Turing Machine

- A Turing machine is said to be universal Turing machine if it can accept:

    - The input data, and
    - An algorithm (description) for computing.
- This is precisely what a general purpose digital computer does. A digital computer accepts a program written in high level language. Thus, a general purpose Turing machine will be called a universal Turing machine if it is powerful enough to simulate the behavior of any digital computer, including any Turing machine itself.
- More precisely, a universal Turing machine can simulate the behavior of an arbitrary Turing machine over any set of input symbols. Thus, it is possible to create a single machine that can be used to compute any computable sequence.

  If this machine is supposed to be supplied with the tape on the beginning of which is written the input string of quintuple separated with some special symbol of some computing machine M, then the universal Turing machine U will compute the same strings as those by M.
- The model of a Universal Turing machine is considered to be a theoretical breakthrough that led to the concept of stored programmer computing device.
- Designing a general purpose Turing machine is a more complex task. Once the transition of Turing machine is defined, the machine is restricted to carrying out one particular type of computation.
- Digital computers, on the other hands, are general purpose machines that cannot be considered equivalent to general purpose digital computers until they are designed to be reprogrammed.
- By modifying our basic model of a Turing machine we can design a universal Turing machine. The modified Turing machine must have a large number of states for stimulating even a simple behavior. We modify our basic model by:
    - Increase the number of read/write heads
    - Increase the number of dimensions of input tape
    - Adding a special purpose memory
- All the above modification in the basic model of a Turing machine will almost speed up the operations of the machine can do.
- A number of ways can be used to explain to show that Turing machines are useful models of real computers. Anything that can be computed by a real computer can also be computed by a Turing machine. A Turing machine, for example can simulate any type of functions used in programming language. Recursion and parameter passing are some typical examples. A Turing machine can also be used to simplify the statements of an algorithm.
- A Turing machine is not very capable of handling it in a given finite amount of time. Also, Turing machines are not designed to receive unbounded input as many real programmers like word processors, operating system, and other system software.

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.

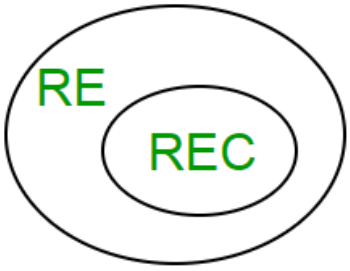


Figure 1

Closure Properties of Recursive Languages

- **Union**: If L1 and If L2 are two recursive languages, their union L1∪L2 will also be recursive because if TM halts for L1 and halts for L2, it will also halt for L1∪L2.
- **Concatenation:** If L1 and If L2 are two recursive languages, their concatenation L1.L2 will also be recursive. For Example:

- L1= $\{a^nb^nc^n|n>=0\}$
- L2= $\{d^me^mf^m|m>=0\}$
- L3= L1.L2
- = $\{a^nb^nc^nd^m\ e^mf^m|m>=0$ and $n>=0\}$ is also recursive.

L1 says n no. of a's followed by n no. of b's followed by n no. of c's. L2 says m no. of d's followed by m no. of e's followed by m no. of f's. Their concatenation first matches no. of a's, b's and c's and then matches no. of d's, e's and f's. So it can be decided by TM.

- **Kleene Closure:** If L1is recursive, its kleene closure L1* will also be recursive.

Unit 6
P Class

Optimization Problem

Optimization problems are those for which the objective is to maximize or minimize some values. For example,

- Finding the minimum number of colors needed to color a given graph.
- Finding the shortest path between two vertices in a graph.

Decision Problem

There are many problems for which the answer is a Yes or a No. These types of problems are known as **decision problems**. For example,

- Whether a given graph can be colored by only 4-colors.
- Finding Hamiltonian cycle in a graph is not a decision problem, whereas checking a graph is Hamiltonian or not is a decision problem.

What is Language?

Every decision problem can have only two answers, yes or no. Hence, a decision problem may belong to a language if it provides an answer 'yes' for a specific input. A language is the totality of inputs for which the answer is Yes

For input size ***n***, if worst-case time complexity of an algorithm is $O(n^k)$, where ***k*** is a constant, the algorithm is a polynomial time algorithm.

Algorithms such as Matrix Chain Multiplication, Single Source Shortest Path, All Pair Shortest Path, Minimum Spanning Tree, etc. run in polynomial time. However there are many problems, such as traveling salesperson, optimal graph coloring, Hamiltonian cycles, finding the longest path in a graph, and satisfying a Boolean formula, for which no polynomial time algorithms is known. These problems belong to an interesting class of problems, called the **NP-Complete** problems, whose status is unknown.

In this context, we can categorize the problems as follows −

P-Class

The class P consists of those problems that are solvable in polynomial time, i.e. these problems can be solved in time $O(n^k)$ in worst-case, where **k** is constant.

These problems are called **tractable**, while others are called **intractable or super polynomial**.

Formally, an algorithm is polynomial time algorithm, if there exists a polynomial $p(n)$ such that the algorithm can solve any instance of size **n** in a time $O(p(n))$.

Problem requiring $\Omega(n^{50})$ time to solve are essentially intractable for large ***n***. Most known polynomial time algorithm run in time $O(n^k)$ for fairly low value of ***k***.

The advantages in considering the class of polynomial-time algorithms is that all reasonable **deterministic single processor model of computation** can be simulated on each other with at most a polynomial slow-d

NP-Class

The class NP consists of those problems that are verifiable in polynomial time. NP is the class of decision problems for which it is easy to check the correctness of a claimed answer, with the aid of a little extra information. Hence, we aren't asking for a way to find a solution, but only to verify that an alleged solution really is correct.

Every problem in this class can be solved in exponential time using exhaustive search.

P versus NP

Every decision problem that is solvable by a deterministic polynomial time algorithm is also solvable by a polynomial time non-deterministic algorithm.

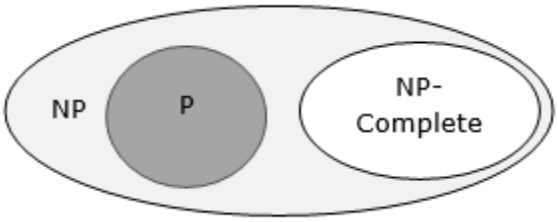
All problems in P can be solved with polynomial time algorithms, whereas all problems in *NP - P* are intractable.

It is not known whether ***P = NP***. However, many problems are known in NP with the property that if they belong to P, then it can be proved that P = NP.

If ***P ≠ NP***, there are problems in NP that are neither in P nor in NP-Complete.

The problem belongs to class **P** if it's easy to find a solution for the problem. The problem belongs to **NP**, if it's easy to check a solution that may have been very tedious to find.

A problem is in the class NPC if it is in NP and is as **hard** as any problem in NP. A problem is **NP-hard** if all problems in NP are polynomial time reducible to it, even though it may not be in NP itself.



If a polynomial time algorithm exists for any of these problems, all problems in NP would be polynomial time solvable. These problems are called **NP-complete**. The phenomenon of NP-completeness is important for both theoretical and practical reasons.

Definition of NP-Completeness

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

- **B** is in NP
- Every **A** in NP is polynomial time reducible to **B**.

If a language satisfies the second property, but not necessarily the first one, the language **B** is known as **NP-Hard**. Informally, a search problem **B** is **NP-Hard** if there exists some **NP-Complete** problem **A** that Turing reduces to **B**.

The problem in NP-Hard cannot be solved in polynomial time, until **P = NP**. If a problem is proved to be NPC, there is no need to waste time on trying to find an efficient algorithm for it. Instead, we can focus on design approximation algorithm.

NP-Complete Problems

Following are some NP-Complete problems, for which no polynomial time algorithm is known.

- Determining whether a graph has a Hamiltonian cycle
- Determining whether a Boolean formula is satisfiable, etc.

NP-Hard Problems

The following problems are NP-Hard

- The circuit-satisfiability problem
- Set Cover
- Vertex Cover
- Travelling Salesman Problem

TSP is NP-Complete

The traveling salesman problem consists of a salesman and a set of cities. The salesman has to visit each one of the cities starting from a certain one and returning to the same city. The challenge of the problem is that the traveling salesman wants to minimize the total length of the trip

Proof

To prove ***TSP is NP-Complete***, first we have to prove that ***TSP belongs to NP***. In TSP, we find a tour and check that the tour contains each vertex once. Then the total cost of the edges of the tour is calculated. Finally, we check if the cost is minimum. This can be completed in polynomial time. Thus ***TSP belongs to NP***.

Secondly, we have to prove that ***TSP is NP-hard***. To prove this, one way is to show that ***Hamiltonian cycle $\leq_p$ TSP*** (as we know that the Hamiltonian cycle problem is NPcomplete).

Assume ***G = (V, E)*** to be an instance of Hamiltonian cycle.

Hence, an instance of TSP is constructed. We create the complete graph ***$G' = (V, E')$***, where

$$E'=\{(i,j):i,j\in V and i\neq j E'=\{(i,j):i,j\in V and i\neq j$$

Thus, the cost function is defined as follows −

$$t(i,j)=\{01if(i,j)\in Eotherwiset(i,j)=\{0if(i,j)\in E1otherwise$$

Now, suppose that a Hamiltonian cycle ***h*** exists in ***G***. It is clear that the cost of each edge in ***h*** is **0** in ***G′*** as each edge belongs to ***E***. Therefore, ***h*** has a cost of **0** in ***G′***. Thus, if graph ***G*** has a Hamiltonian cycle, then graph ***G′*** has a tour of **0** cost.

Conversely, we assume that ***G′*** has a tour ***h′*** of cost at most **0**. The cost of edges in ***E′*** are **0** and **1** by definition. Hence, each edge must have a cost of **0**as the cost of ***h′*** is **0**. We therefore conclude that ***h′*** contains only edges in ***E***.

We have thus proven that ***G*** has a Hamiltonian cycle, if and only if ***G′*** has a tour of cost at most **0**. TSP is NP-complete.