Turing Machines

Until right now we have seen PDA or FA. The main difference between PDA, FA and Turing machines is that PDA and FA move in only one direction while turning machine can move in both the directions. Turning machine can read and write a symbol on the tape.

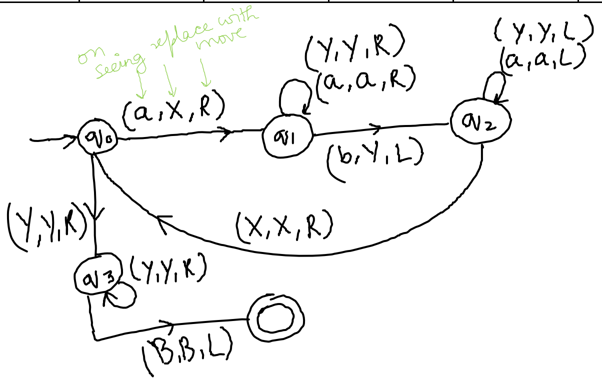
Turning machine:

1. It can move left and right
2. It can do both read and write operations on the tape
3. It may not have all the configurations from a state and still be deterministic
4. If a string is not accepted by a turing machine, the machine will halt in a state other than the final state
5. If a string is accepted by a turing machine, the machine will halt in the final state
6. Turing machine is a hypothetical infinite tape with cells.

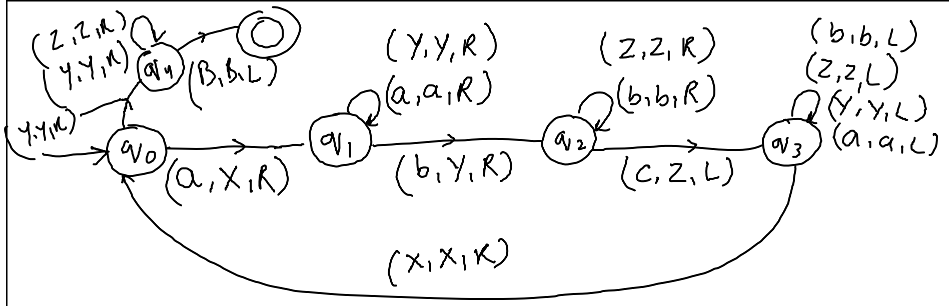
Let's see an example:

Check whether the following string is accepted or not: B = Blank

… B	B	a	a	a	b	b	b	B	B …



Let's see one more example: L = {$a^nb^nc^n$ | n>=1}, make a turning machine for this one.



Turning machine: A transducer.

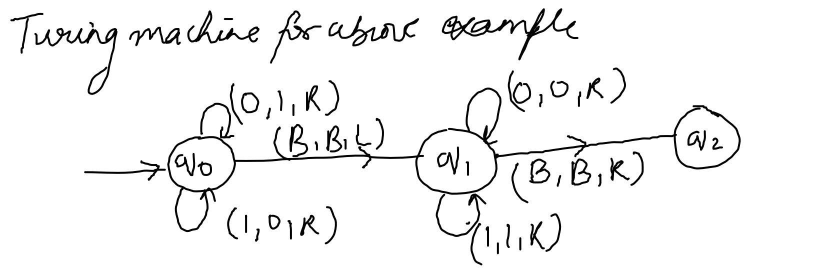
FA and PDA are considered as acceptors. This means that when we send a symbol to these machines, they will tell us if they accept or reject the string or whether the input is present or absent in the language. On the other hand:

Turning machine can act as an acceptor and as a transducer. A transducer converts an input to an output. So turning machines can convert a given input to give an output too. Because turning machine has the capacity to read from the tape and write on it too, it has the capacity to change the symbols on the tape, meaning changing the input to some other output. Let's take an example.

Example 1: TM to find one's complement of a binary string.

Note: It is a courtesy to leave the output at the start of the final modified string. Accha nahi lagta ki Blank pe chhod diya. Lol.

Here the turning machine is not an acceptor of string, therefore there is no need of a final state. It can halt whenever the work has been done.



Let's see another example

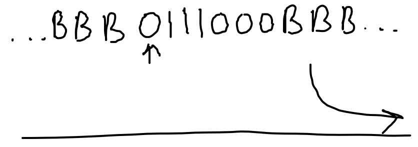
Example 2: Turing machine to find the 2's complement of a given binary string

Logic: From right, if zeros keep them as zeros, 1^{st}one leave it as it is, 1's complement the remaining symbols on the left of 1^{st} 1.

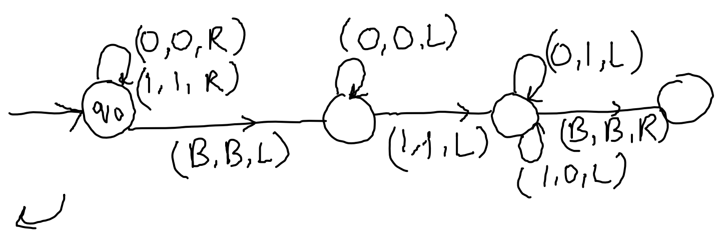
String : 0111000

One's : 1000111

Two's: 1001000



Let's take another example

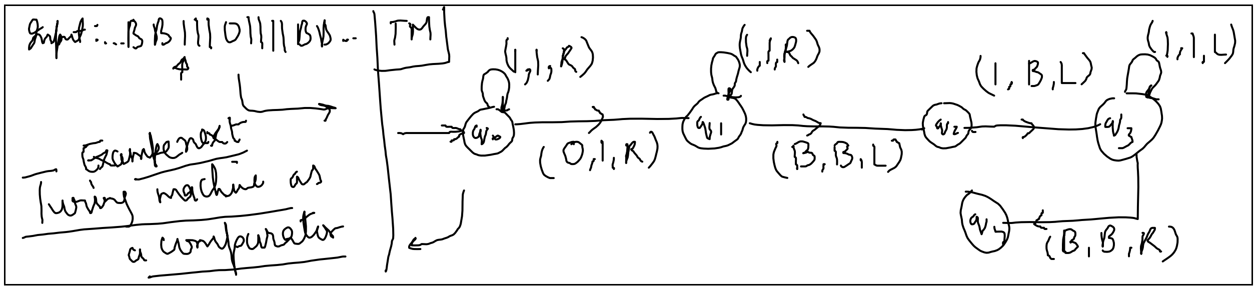


Example 3: Turning machine as an adder

Let’s use unary representation of the number.

3 = 111, 4 = 1111, 5 = 11111 and so on.

String will be given like: 11101111 (for adding 3 and 4) and we want in the end 1111111 (7) as the answer. Let’s do this.

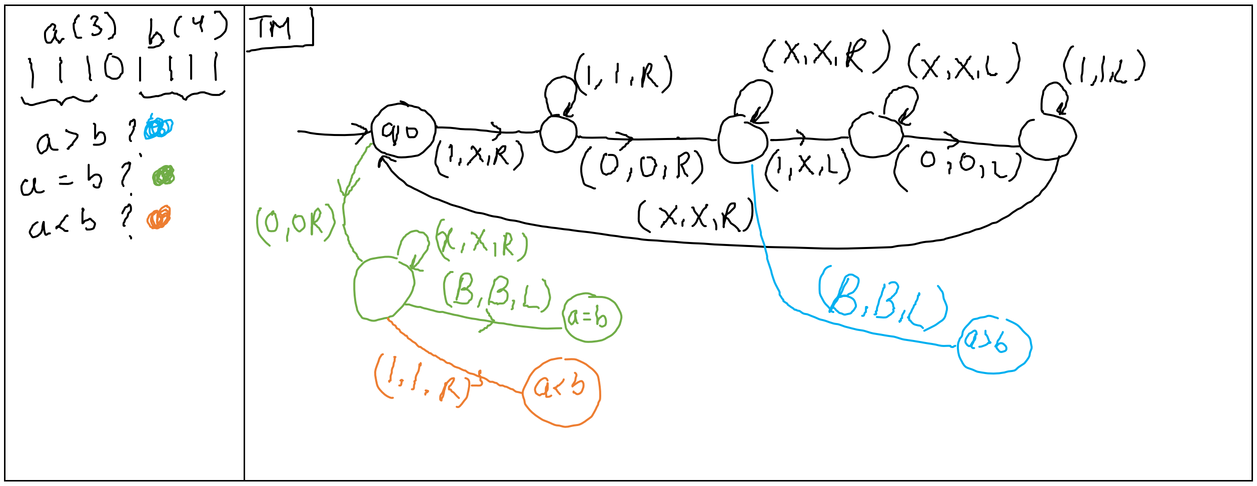


Example 4: Turning machine as a comparator

a=111 (number 3) , b=1111 (number 4). The turning machine should give me three things.

1. Whether a and b are equal (should halt in a state that indicates a is equal to b)
2. Whether a is greater than b (should halt in a state that indicates a >b)
3. Whether a is smaller than b (should halt in a state that indicates a < b)

Therefore, depending on the input our turning machine should always halt in one of the three states. Let’s see how can we do that.



Note: addition and comparison are the basis operations for any mathematical operation. So by having a cascade of Turing machines you can represent functions like log, exponents, subtraction, division, multiplication by just addition and comparison operations. Therefore, Turing machine is mathematically complete. This means that given any mathematical function, *Turing machine will be able to compute it because it is able to do addition and comparison. So* ***Turing machine can perform any mathematical function.***

STANDARD TURING MACHINE

Let's define the standard turing machine. "Standard" because there have been various modifications to Turning machines. However, all of them are equivalent to each other.

A standard Turning machine is a sept-tuple defined as

$M = \{ Q, \Sigma, \Gamma, \delta, q_0, B, F \}$

Q: Set of states	δ**:** transition function	$F \lesssim Q$: set of final states
Σ**:** input alphabet	$q_0 \in Q$: initial state	
Γ**:** tape alphabet	$B \in \Gamma$: used to represent blank	

δ: $(Q \times \Gamma) \rightarrow (Q \times \Gamma \times \{L, R\})$ [this is a partial function, because there are dead configuration, so for some of the states and inputs there is no determinism. We don't define for every symbol and every state]

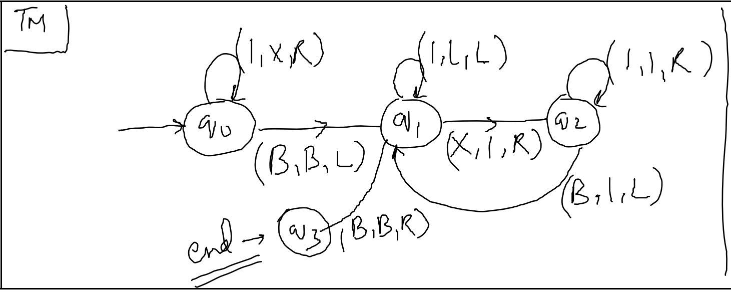
Γ is the tape alphabet which consists of all the symbols that can be written on a tape. Is the input alphabet is {a, b} then let's say tape alphabet = {a, b, X, Y, B}. Input alphabet is a subset of tape alphabet.

If you want to visualize Turing machines, then take FA add two stacks to it (with one stack it is PDA). If there are two stacks, I can use that machine with a stack or a queue and can do anything. That is a Turing machine. Those two stacks can be merged and seen as a tape.

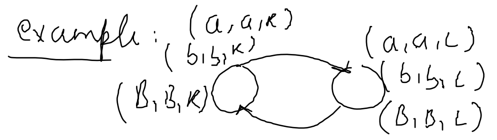
Notes:

1. Tape is unbounded, so any number of left and right moves are possible
2. It is deterministic, i.e., at most one move for each configuration. Looking at the same configuration you will never see two transitions in a Turning machine.
3. No special input or output file. Input and output remain on the tape.

Example 5: Turing machine as a copier.

i/p : BBB111BBB o/p: BBB111111BBB	TM  (1,X,R) (1,1,L) (1,1,R) q0 q1 q2 q3 (B,B,L) (X,1,R) (B,1,L) (B,B,R) end →	can be used to multiply two numbers 110111 ⟸ 111111 (2) (3)

Non-halting Turing machine: A Turing machine which doesn't halt. It is almost like being in an infinite loop. This kind of behavior is not possible in unidirectional automata like FA or PDA.



This is called halting problem in Turing machines.

Turing Thesis

Any computation that can be carried out by mechanical means can be performed by a Turing machine. This means Turing machine can solve every problem solved by the said mechanical means.

Some arguments why Turing thesis is accepted as definition of mechanical computation or computer:

1. Anything that can be done by existing digital computer can also be done by Turing machine.
2. No one has yet been able to suggest a problem, solvable by what we intuitively consider an algorithm, for which a Turing machine program cannot be written.
3. Alternative models have been proposed for mechanical computation but none of them are more powerful than the Turing machine model.

Some modifications to standard Turing machines:

Power of TM: Number of languages a TM can accept.

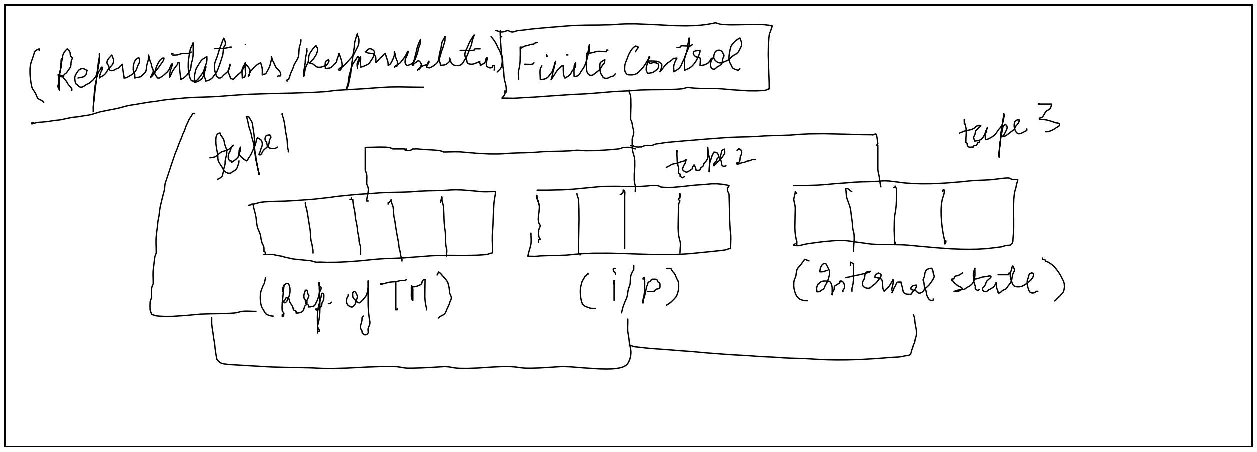
It is seen that the number of languages accepted by a standard TM and modified ones will be same. But, the time taken to accept a language may be different. Which means the number of steps required may increase or decrease.

But changes don't increase the power. Here we are discussing the TM modifications which do not affect the power of the TM.

1. **TM with stay option**: The TM need not move to the left or right, it can also stay in a particular state. δ: (Q x Γ) -> (Q x Γ x {L, R, **S**})
2. **TM with semi-infinite tape**: The tape is not infinite on both the side.
3. **Offline TM:** Input in separate file which is read-only. So finite control of TM can read the symbol from the input and write on the TM tape. If modifications required in the input, copy the input on the tape and then do the modifications on the tape.
4. **Multi-tape TM:** δ: $(Q \times \Gamma^n) \rightarrow (Q \times \Gamma^n \times \{L, R\}^n)$ | n = number of tapes we have.
5. **Jumping TM:** Instead of moving one step ahead it can go any number of steps to the right. δ: (Q x Γ) -> (Q x Γ x {L, R} x {n}) | n = steps
6. **Non-erasing TM:** Standard TM can change the input to a blank. In non-erasing TM you can't do that.
7. **Always writing TM:** In standard TM we have liberty to leave the symbol as it is. Here you are supposed to definitely change the symbol machine has read to some other symbol.
8. **Multidimensional TM:** δ: (Q x Γ) -> (Q x Γ x {L, R, U, D}). In our tape, instead of having 2 dimensions you can also go up and down which is respectively U and D.
9. **Multihead TM:** The finite control can look at various places simultaneously.
10. **Automata with a queue:** There is a finite automata with queue.
11. **TM with only 3 states**
12. **Multitape Tm with stay option and at most 2 states**
13. **Non-deterministic TM:** δ: $(2^{Q \times \Gamma}) \rightarrow (2^{Q \times \Gamma} \times \{L, R,\})$
14. **A NPDA with two independent stacks:** δ: $(Q \times (\Sigma \cup \varepsilon) \times \Gamma \times \Gamma) \rightarrow 2^{Q \times \Gamma^* \times \Gamma^*}$

UNIVERSAL TURING MACHINE

A universal Turing machine has a finite control but it is going to have 3 tapes. So it is a multi-tape Turing machine. Let's see what these three tapes do.



The first tape is the entire representation of the Turing machine which UTM is trying to accommodate.

The second tape is the input that the Turing machine should work on.

The third tape is the manager of the internal states.

Example: Let's say that you want to add two numbers, 2 and 3.

The input will be 110111 and that will be saved on the second tape of the universal Turing machine. The representation of the turing machine which will only add two numbers (not the UTM) will be saved in the first tape. The internal state governed by the transition function will be saved in the third tape. The first symbol 1 (from 110111) will be read from the second tape. The third tape will have the initial state q_0 so according to the transition function $\delta(q_0, 1) = (q_2, x, R)$, the symbol x will be written in the second tape and the third tape will have the state q_2 now. This transition is governed in the first tape which holds the representation of the Turing machine which would add two numbers.

The representation of the Turing machine that does one thing has to be saved in tape 1. This sounds like a mystery, so how are we going to do that?

We assume that the Turing machine has

Q = $\{q_0, q_1, q_2, q_3, q_4, ...\}$ (set of all states)

Γ = $\{a_0, a_1, a_2, a_3, a_4, \ldots\}$ (tape alphabet)

Then you have to encode all the symbols, let's say the encoding is as follows:

$q_1 \rightarrow 1$, $q_2 \rightarrow 11$, $q_3 \rightarrow 111$, $q_4 \rightarrow 1111$, similarly

$a_1 \rightarrow 1$, $a_2 \rightarrow 11$, $a_3 \rightarrow 111$, $a_4 \rightarrow 1111$

R → 11, L → 1

'0' → separator

So for a transition like $\underbrace{\delta(\overset{1\,0\,11}{\overline{q_1}, \overline{a_1}}) = (\overset{11011011}{\overline{q_2}, \overline{a_2}, \overline{R}})}$ the binary representation will be $\overset{\text{initial}}{\downarrow 0}\underbrace{101011011011}\overset{\text{final}}{\downarrow 0}$

Therefore, every Turing machine can be encoded as a string of 0's and 1's. (very important). These strings of 0's and 1's for all transitions possible will be kept in the Tape 1.

So if I'm going to represent a Turing machine in terms of 0's and 1's then I can say that any possible Turing machine will be an element of: $\Sigma^*, \Sigma \in \{0,1\}$. But this does not mean that every element of this infinite set is going to be a Turing machine. Please know the following statements.

Statement 1: Every Turing machine can be represented as a string of 0's and 1's

Statement 2: Not every string of 0's and 1's is a Turing machine.

LINEAR BOUNDED AUTOMATA

We know that Turing machines have been modified. All of these Turing machines that we saw were equivalent to each other, i.e., the languages accepted by all of the modified Turing machines is same. Let's take some more modifications.

1. A non-deterministic Turing machine with a tape that acts as a stack: this will be equivalent to the PDA.
2. A Turing machine with a finite tape: This will act as our Finite Automata.

Here, we are essentially reducing the power of a TM by putting some limitations on the tape it can operate on. Let's see one more modification:

A Turing machine with the size of the tape which is equal to the size of the input.

Now we are putting a boundary on the tape size. This particular result of putting a boundary gives birth to a new machine which is called *Linear Bounded Automata.*

A linear bounded automaton (LBA) is powerful than FA and PDA. Remember the example that $L = \{a^n b^n c^n \mid n >= 0\}$ is a language which can be accepted by LBA but not a PDA. For LBA to be more powerful than the PDA, it should accept all the languages accepted by PDA. This can be proved by making the tape of LBA as the stack of PDA and a PDA will not require the stack size more than the input to accept a string. Therefore, every PDA can have an equivalent LBA but the inverse is not true. Thus, LBA is powerful than PDA.

The boundary indicators < and > in the LBA tape are called endmarkers.

So the power dynamics are:

- FA < PDA < LBA < TM

The power of the deterministic and non-deterministic machines in:

- FA will be equal
- PDA will not be equal
- LBA *we don't know as no one has found it out yet.*
- TM will be equal

Let' see some general examples of languages which can be accepted by a LBA but not by a PDA.

$$L = \{a^n b^n c^n d^n_{\dots}, n \geq 1\}$$

$$L = \{a^{n!}, n \geq 0\}$$

$$L = \{a^n : n = m^2, m \geq 1\}$$

$$L = \{a^n : n\ is\ a\ prime\}$$

$$L = \{a^n : n\ is\ not\ a\ prime\}$$

$$L = \{ww : w \in \{a,b\}^+\}$$

$$L = \{w^n ; w \in \{a,z\}^+, n \geq 1\}$$

$$L = \{www^R : w \in \{a,b\}^+\}$$

These are some standard examples of languages which are accepted by LBA apart from all the languages which are accepted by a PDA. So what is the set of all languages accepted by Turing machines?

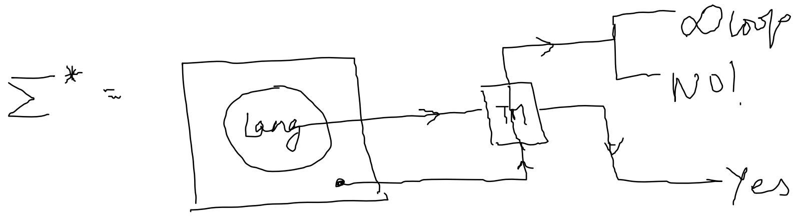
LANGUAGE ACCEPTED BY TURING MACHINES

We saw Regular Languages are accepted by FA and Context Free Languages are accepted by PDA. The languages accepted by a TM are called *Recursively Enumerable Languages.*

Since languages accepted by PDA are also accepted by TM, therefore CFLs are Recursively Enumerable.

Turing machine cannot accept ε but a PDA does. So when I say CFLs are a subset of RELs (recursively enumerable languages) then I am talking about the ones which don't have ε. Also ε don't make difference.

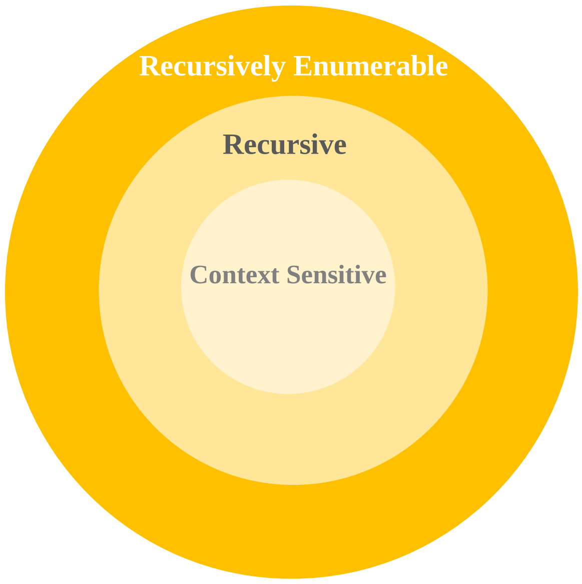
Now when a string is the language the Turing machine will accept, say Yes and halt in a finite state. If the string is not in the language the TM will say No or *it will enter an infinite loop.*



So whenever the TM enters an infinite loop, we don't know whether to wait or to halt the Turing machine. Now, I don't want such a dilemma. I want a TM that says YES or NO! So, if I restrict a Turing Machine in such a way that it says only Yes or No, such a Turing machine is called *HALTING TURING MACHINE.* Therefore, a HTM will always halt no matter the language given to it.

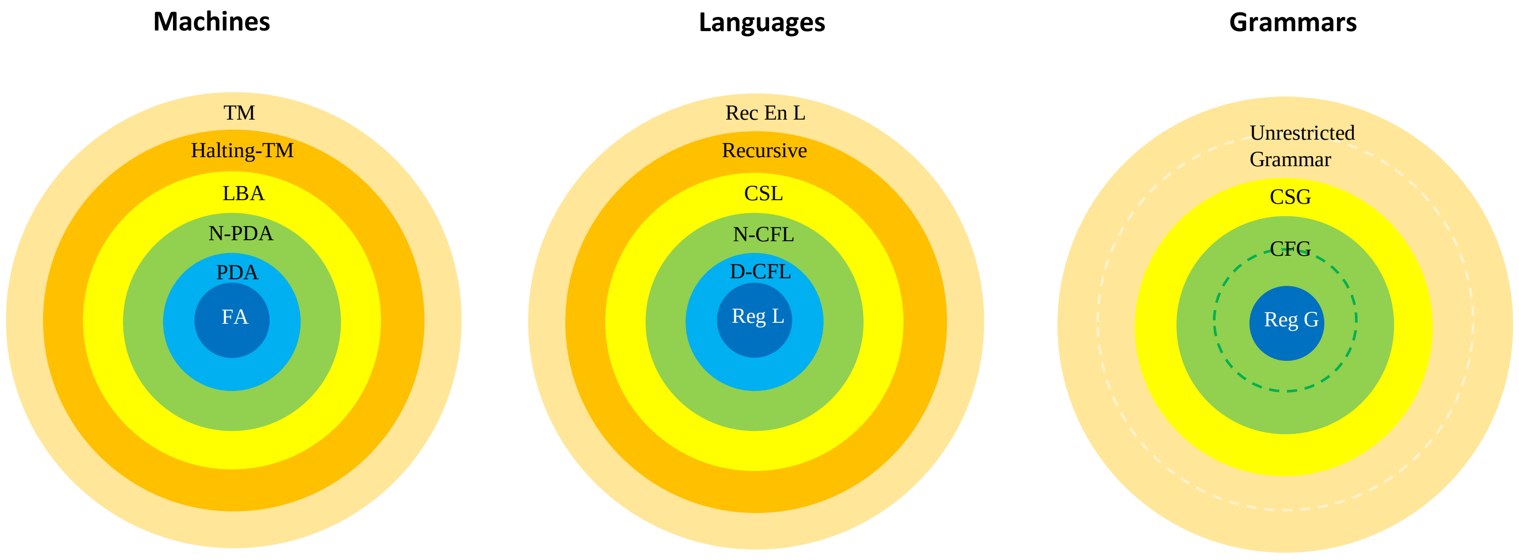
*Note: The languages accepted by **Turing Machines** are called **Recursively Enumerable Languages** and the languages accepted by **Halting Turing Machines** are called **Recursive Languages**. Recursive languages are a subset of Recursively Enumerable Languages*

*Moreover, the languages accepted by **Linear Bounded Automata** are **called Context Sensitive Languages. LBA is a halting Turing machine**. The relationship between these languages is:*



Now let’s see what different types of machines we have seen, what are the languages associated with them and what are the grammars associated with them.

THE BIG PICTURE



Let’s see them one by one and learn about their properties. By the way:

- Unrestricted grammars are also called Type 0 Grammars
- Context-sensitive grammars are also called Type 1 Grammars
- Context-free grammars are also called Type 2 Grammars
- Regular grammars are also called Type 3 Grammars

UNRESTRICTED GRAMMAR

The languages corresponding to TM are Recursively Enumerable Languages and the generator of this language is the Unrestricted Grammar.

A grammar is called <u>unrestricted</u> if all the productions are of the form:

$$u \to v, u \in (V \cup T)^{+}, v \in (V \cup T)^{*}$$

This means the LHS of the production rule can have anything except ε meaning the null string should not derive any string. The right hand side is completely unrestricted.

Example: What language does the following unrestricted grammar generates?

$S \to S_1B$

$S_1 \to aS_1b$

$bB \to bbbB$

$aS_1b \to aa$

$B \to \lambda$

CONTEXT SENSITIVE GRAMMAR

A grammar is said to be context sensitive if all the production formulas are of the form:

$$x \to y: x, y \in (V \cup T)^{+}$$

and

$$|x| \leq |y|$$

This means that the LHS and RHS can't have ε and there will never be a contraction from LHS to RHS. This means at every step of the derivation the string length either remains same or increases but *never decreases.*

Now, it is called context-sensitive (let's understand with example):

In context free we see: A → a (here the variable A can be replaced by a terminal 'a' independent of the 'context' or extra information around A)

In context sensitive: aAb → aab (here we are not replacing the variable A independently to 'a'. The A will only be replaced with 'a' terminal *iff* A occurs with 'a' and 'b' as prefix and suffix respectively so A → a is not a production rule here)

Example: What is the language generated by the following CSG.

$S \to abc/aAbc$

$Ab \to bA$

derivation : ① $S \Rightarrow \boxed{abc} \to \underline{\underline{a^1b^1c^1}}$

② $S \Rightarrow aAbc$

$\Rightarrow abAc$

$\Rightarrow abBbcc$

③ $S \Rightarrow aAbc$

$\Rightarrow abAc$

$\Rightarrow abBbcc$

$\Rightarrow aBbbcc$

$\Rightarrow$ [illegible]

$\Rightarrow aBbbcc$

$\Rightarrow \boxed{aabbcc} = \boxed{a^2b^2c^2}$

$\Rightarrow aaAbbcc$

$\Rightarrow aabAbcc$

$\Rightarrow aabbAcc$

$\Rightarrow aabbBbccc$

$\Rightarrow aabBbbccc$

$\Rightarrow aaBbbbccc$

$\Rightarrow aaabbbccc = \boxed{a^3b^3c^3}$

Ac → Bbcc

bB → Bb

aB → aa/aaA

Language. $\{a^n b^n c^n \mid n \geq 1\}$

IMPORTANT THEOREM ON RECURSIVE AND RE LANGUAGES

Theorem:

1. If a language L and its complement L` are both recursively enumerable, then both languages are recursive.
2. If L is recursive, then L` is also recursive and consequently both are recursively enumerable.

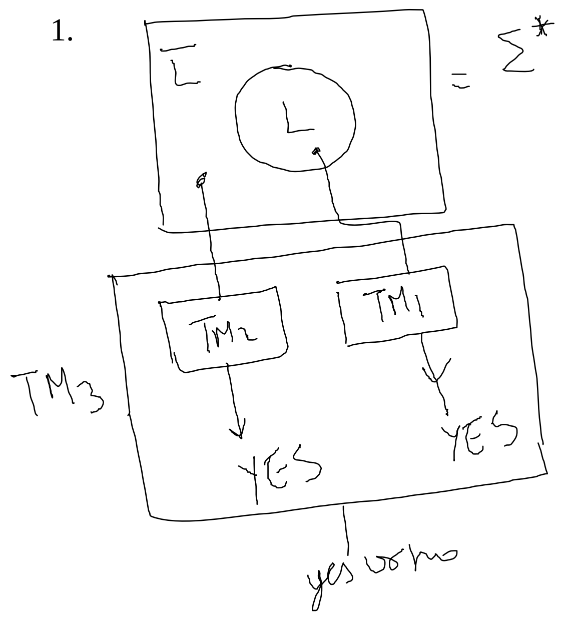
The difference between TM and HTM we already know. The *membership algorithm* also asks the same: *Given a string and a language, whether the string belongs to a language or not?* If there is an algorithm that can answer the above question that is called a membership algorithm.

So, ***for Recursively Enumerable languages, the membership algorithm does not exist***.

But, ***for the Recursive Languages, the membership algorithm exists and answers the membership question with a YES or NO!***

Let's understand the theorems.

1.

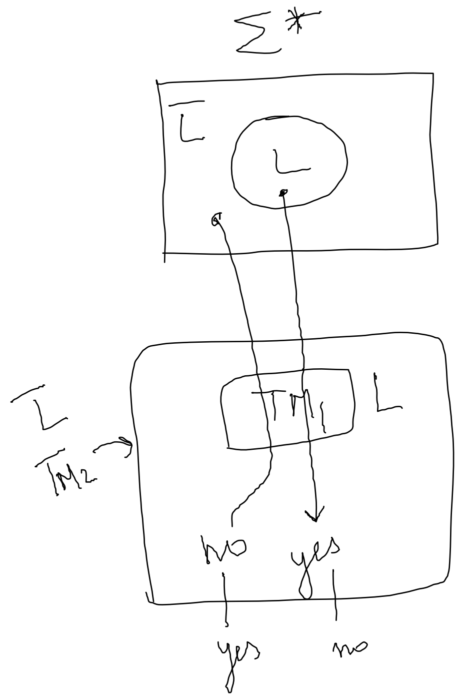


The theorem says *that if the language L and its complement L` both are recursively innumerable*. This means that the language L and L`, both can be represented by two separate Turing machines. Now if a language is accepted by a Turing machine it is always going to halt.

Let's create a Turing machine consisting of both TM1 and TM2 which accept L and L` languages respectively (TM3). Now this Turing machine can take any string from the Σ* and come to a YES or NO outcome with a halt.

Thus *L and L` are both recursive.*

2. Let's take a look at the second point of the theorem.



The second statement says that *if L is recursive, then L` is also recursive*. This means that there is a HTL for L and if we create a HTL which complements the output of TM1, then this TM2(diagram) will represent an HTL for the language L`.

Thus *consequently both are recursively enumerable* because RE just wants a Turing machine, it doesn’t care if it halts or not.