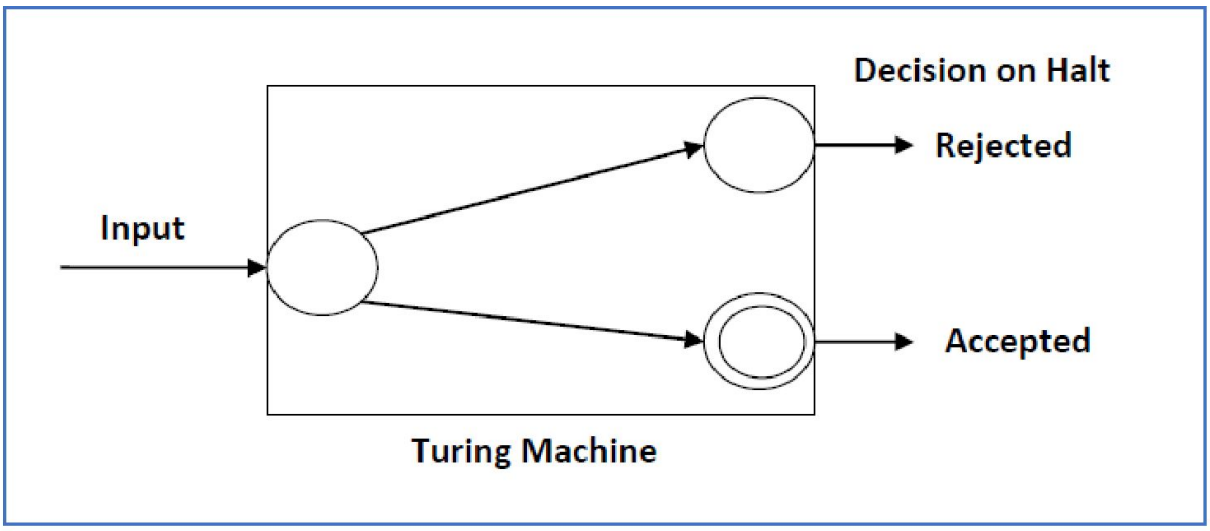
UNIT – VI: Computability

Decidable Problem

A Problem or language(L) is said to Decidable or Solvable if there exists a Turing machine that halts on every input strings, that means the Turing machine will accept and Halt on all strings which belongs to the language and Turing Machine will reject and Halt on all strings which does not belongs to that language.
(or)
A language is called **Decidable** or **Recursive** if there is a Turing machine which accepts and halts on every input string **w**. Every decidable language is Turing-Acceptable.



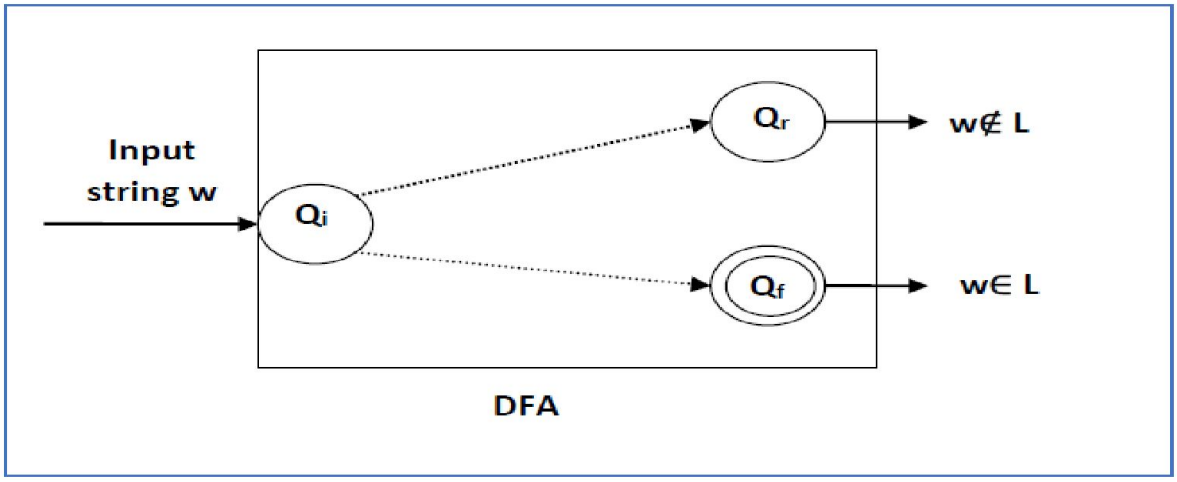
Example 1

Find out whether the following problem is decidable or not:
Is a number 'm' prime?
Solution
Prime numbers = {2, 3, 5, 7, 11, 13, ……..}

Divide the number '**m**' by all the numbers between '2' and '$\sqrt{}$**m**' starting from '2'. If any of these numbers produce a remainder zero, then it goes to the "Rejected state", otherwise it goes to the "Accepted state". So, here the answer could be made by 'Yes' or 'No'.
Hence, it is a decidable problem.

Example 2

Given a regular language **L** and string **w**, how can we check if **w**$\in$ **L**?
Solution
Take the DFA that accepts **L** and check if **w** is accepted



Some more decidable problems are:
1. Does DFA accept the empty language?
2. Is $L_1 \cap L_2 = \emptyset$ for regular sets?

Note:
1. If a language **L** is decidable, then its complement **L'** is also decidable.
2. If a language is decidable, then there is an enumerator(Turing Machine) for it.

Semi-Decidable Problem or Partially Decidable:

A Problem or language(L) is said to Semi-Decidable or Semi-Solvable if there exists a Turing machine that may halts or may not halt on every input strings, that means the Turing machine will accept and Halt on all strings which belongs to the language and Turing Machine will Not Halt on all strings which does not belongs to that language.

Un-Decidable Problem

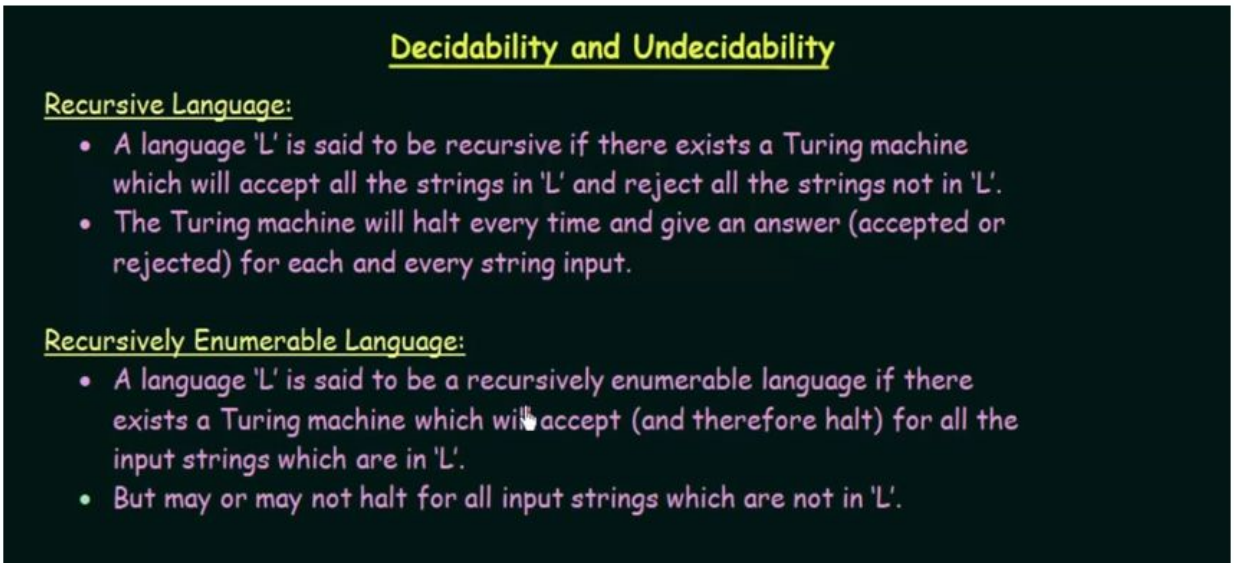
A Problem or language (L) is said to Un-Decidable or Unsolvable if we cannot construct a Turing Machine.

Note: Semi Decidable problems are also Un Decidable.

Examples:

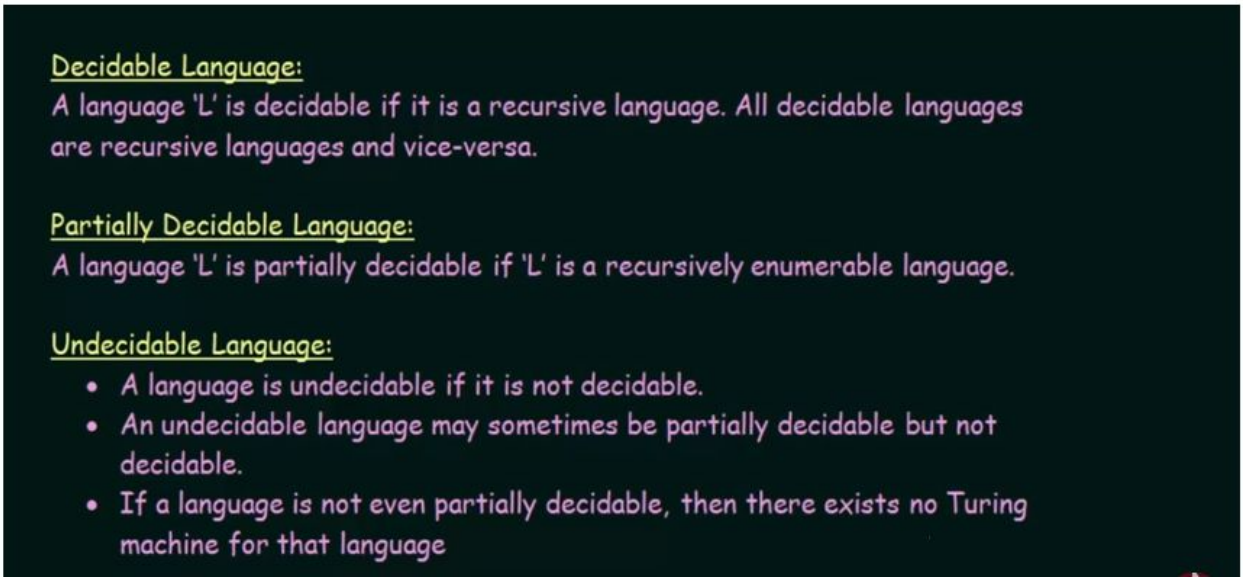
- The halting problem of Turing machine
- The mortality problem
- The mortal matrix problem
- The Post correspondence problem, etc.

Recursive & Recursive Enumerable languages

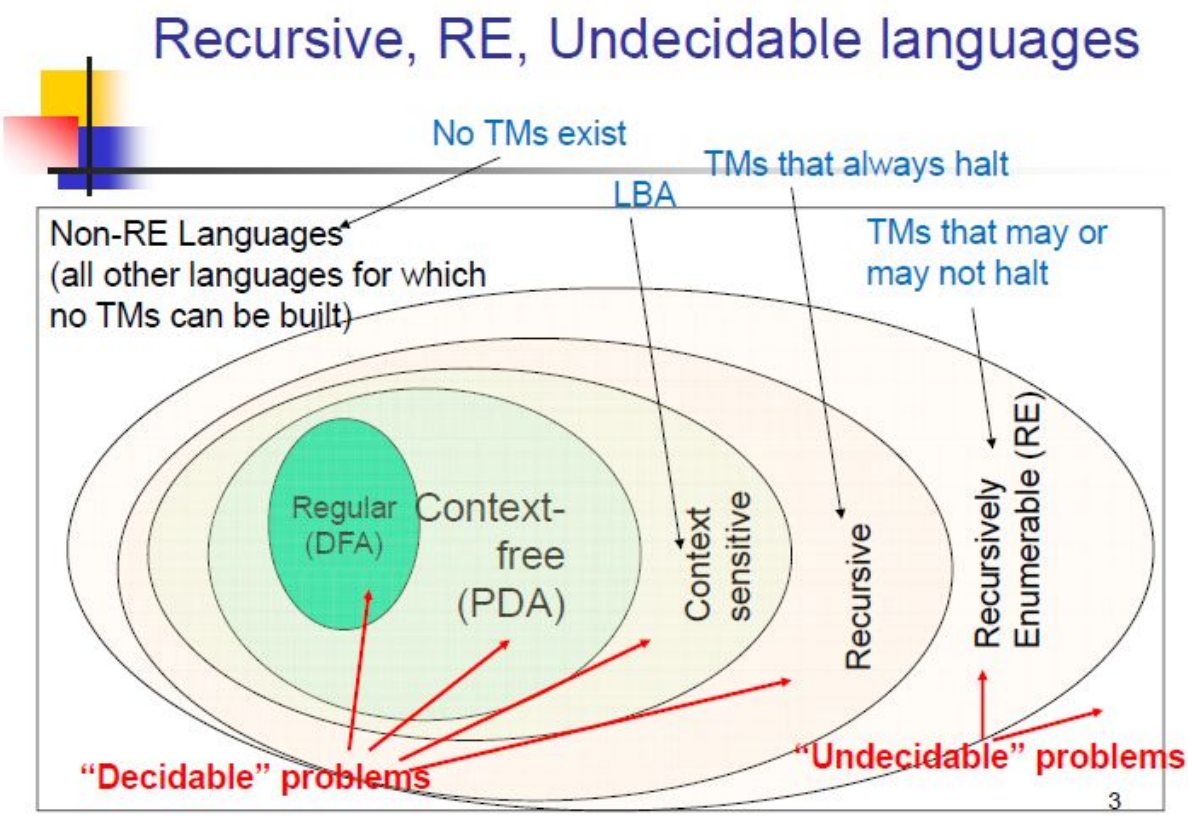


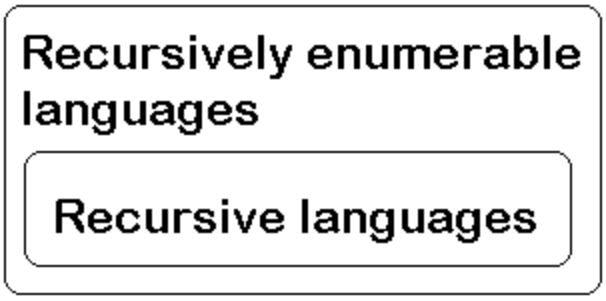
Non Recursively Languages:

A language L is said to be Non Recursively Language if we cannot construct a Turing machine.



Recursive Language	TM will always Halt
Recursively Enumerable Language	TM will halt sometimes & may not halt sometimes
Decidable Language	Recursive Language
Partially Decidable Language	Recursively Enumerable Language
UNDECIDABLE	No TM for that language



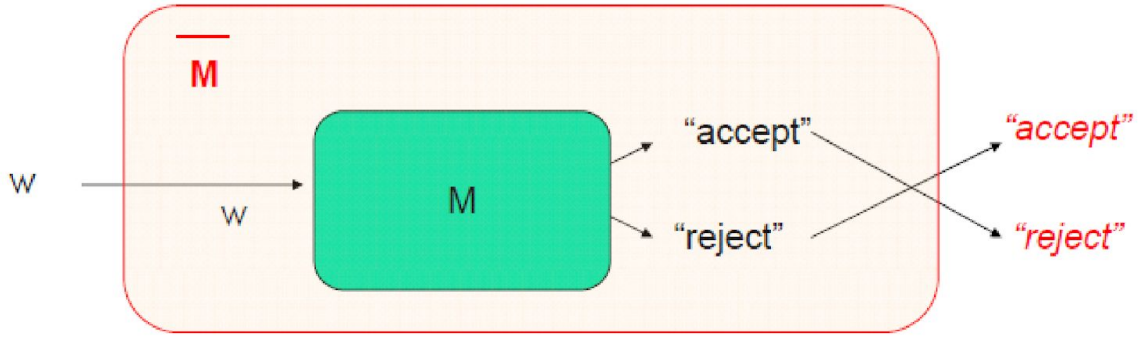


Note:Every Recursive Language is Recursive enumerable language but Vice-versa is not True.

Closure properties of Recursive Languages:

1) Recursive Languages are closed under complementation

If L is Recursive, L^1 is also Recursive

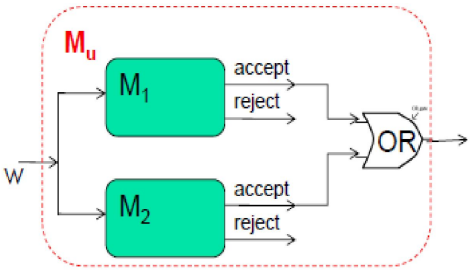


2) Recursive Languages are closed under Union

Let Mu = TM for L1 U L2

Mu construction:

1. Make 2-tapes and copy input w on both tapes
2. Simulate M1 tape 1
3. Simulate M2 on tape 2
4. If either M1 or M2 accepts, then Mu accepts
5. Otherwise, Mu rejects.

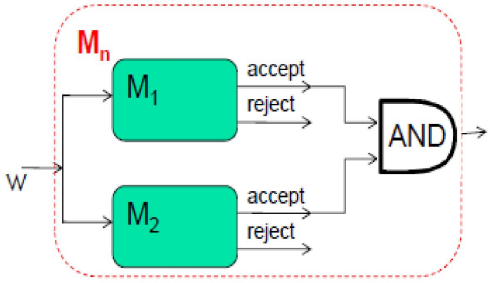


3) Recursive Languages are closed under Intersection

Let Mn = TM for L1 L2

Mn construction:

1. Make 2-tapes and copy input w on both tapes
2. Simulate M1 tape 1
3. Simulate M2 on tape 2
4. If either M1 or M2 accepts, then M_n accepts
5. Otherwise, M_n rejects.



4) Recursive languages are also closed under Concatenation.

5) Recursive languages are also closed under Kleene closure (star operator).

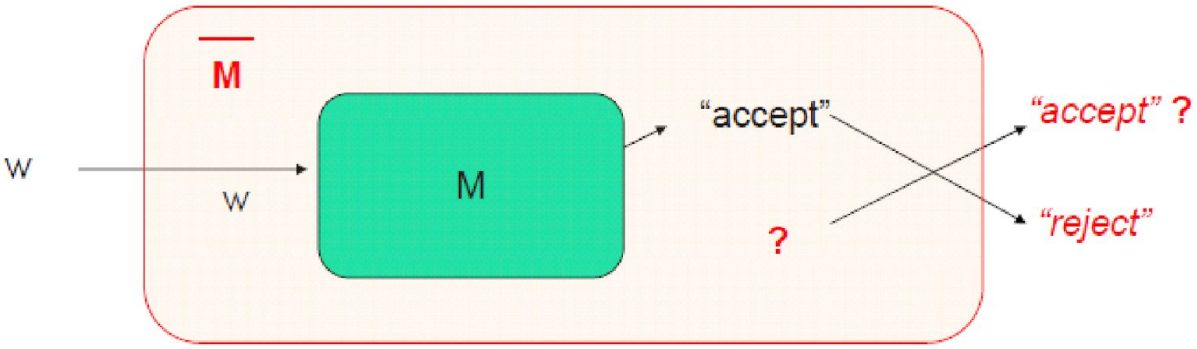
6) Recursive languages are also closed under Homomorphism, and inverse homomorphism.

Closure properties of Recursive Enumerable Languages:

1) **Recursively Enumerable Languages are not closed under Complementation.**

If L is RE, L^1 need not be RE.

2)



Recursive Enumerable languages are also closed under Union.

3) Recursive Enumerable languages are also closed intersection

4) Recursive Enumerable languages are also closed under Kleene closure (star operator).

5) Recursive Enumerable languages are also closed concatenation

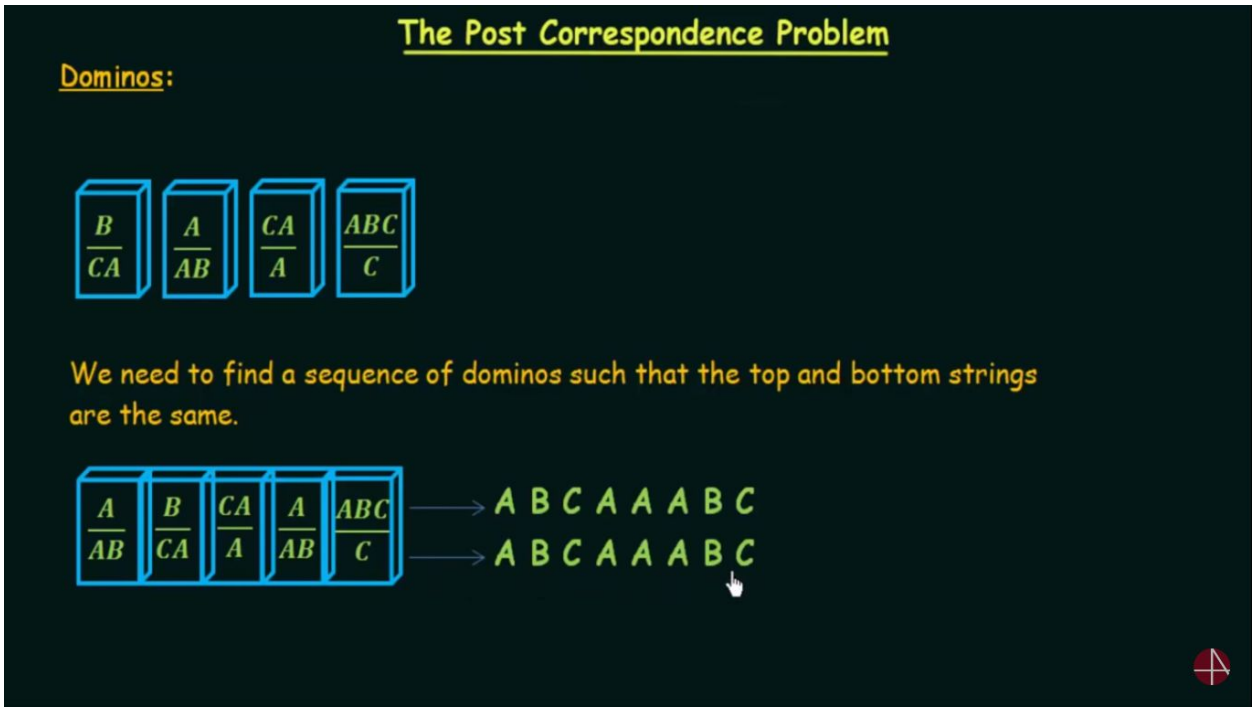
Post Correspondence problem (PCP)

The Post Correspondence Problem (PCP), introduced by Emil Post in 1946, is an undecidable decision problem.

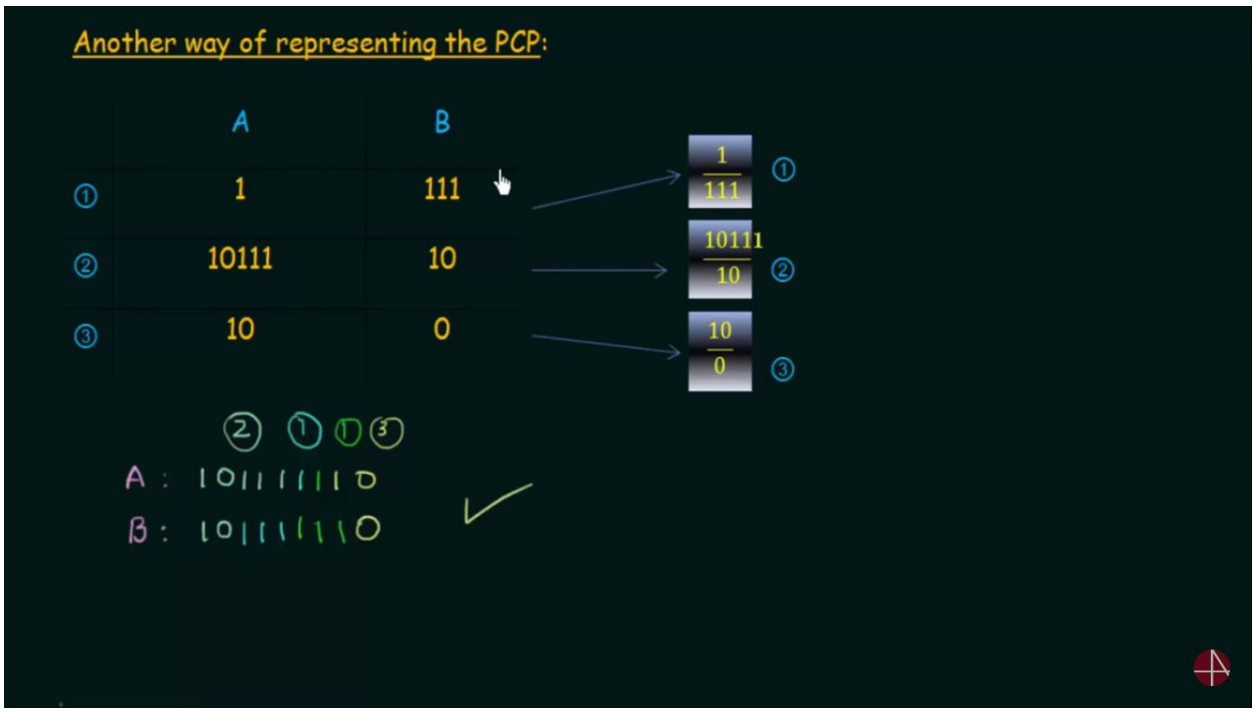
A PCP consists of two lists of string over some alphabet Σ; the two lists must be of equal length. Generally A=w1,w2, w3, …….wk and B= x1, x2 ,x3, ……xk for some integer k. For each i , the pair (wi , xi) is said to be a corresponding pair. We say this instances of PCP has a solution, if

there is a sequence of one or more integers i1, i2,……., im that, when interpreted as indexes for strings in the A and B lists, yield the same string. That is, wi1 wi2 ……. wim = xi1 xi2 ……. xim . We say the sequence i1, i2,……., im is a solution to this instance of PCP

Post Correspondence Problem (PCP) can be learned from dominos game.



Example 1:

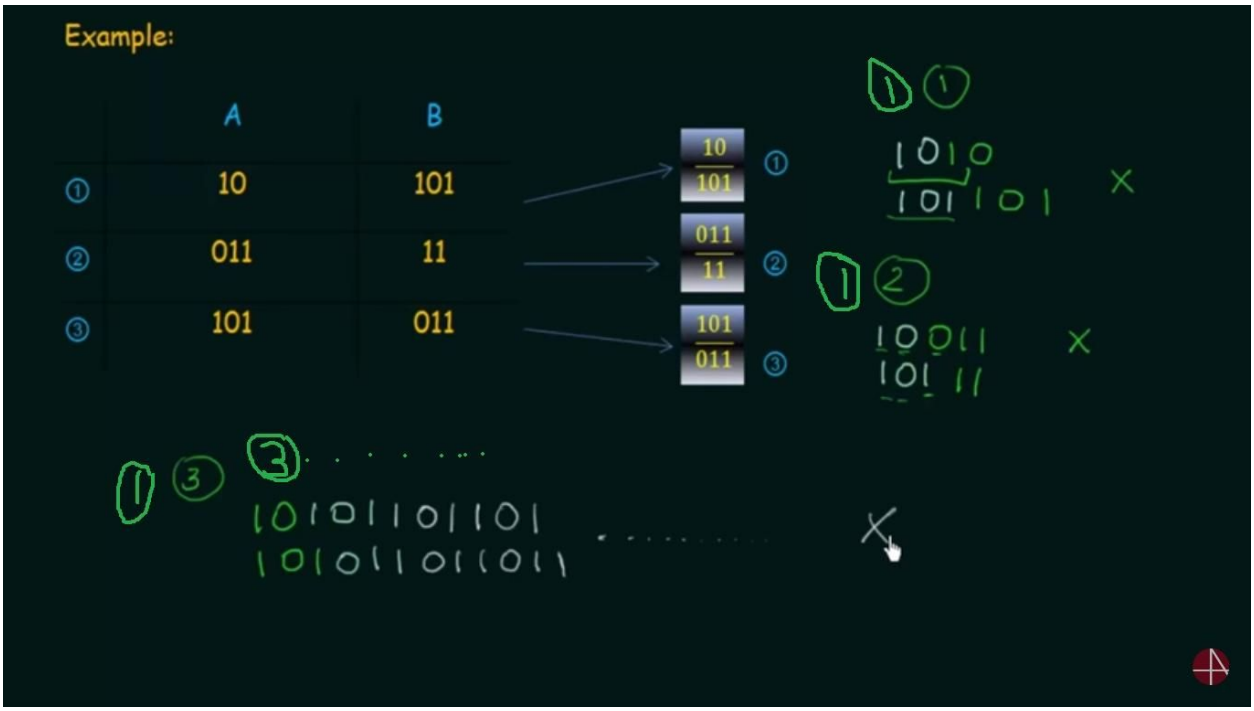


2	10111	10
3	10	0

This PCP instance has a solution: 2, 1, 1, 3:

$$w_2w_1w_1w_3 = x_2x_1x_1x_3 = 101111110$$

Example 2:



Consider Lists A and B

	List A	List B
i	w_i	x_i
1	10	101

List Order :

1 3 3...

In Example 1 we are having a sequence so the PCP has a solution. In example 2 we are not having a sequence so the PCP has no solution. That means we cannot write a generic algorithm to find sequence or solution for every PCP instance. Hence PCP is said to be Undecidible.

The Halting problem

Given a program/algorithm will ever halt or not?

Halting means that the program on certain input will accept it and halt or reject it and halt and it would never goes into an infinite loop. Basically halting means terminating. So can we have an algorithm that will tell that the given program will halt or not. In terms of Turing machine, will it terminate when run on some machine with some particular given input string.
The answer in no we cannot design a generalised algorithm which can appropriately say that given a program will ever halt or not?

It is only possible for us to run the program and check whether it halts or not.
We can refrain the halting problem question in such a way also: Given a program written in some programming language(c/c++/java) will it ever get into an infinite loop(loop never stops) or will it always terminate(halt)?

This is an undecidable problem because we cannot have an algorithm which will tell us whether a given program will halt or not in a generalised way i.e by having specific program/algorithm.In general we can't always know that's why we can't have a general algorithm.The best possible way is to run the program and see whether it halts or not.In this way for many programs we can see that it will sometimes loop and always halt.

Proof by Contradiction –

Problem statement: Can we design a machine which if given a program can find out if that program will always halt or not halt on a particular input?
Solution: Let us assume that we can design that kind of machine called as HM(P, I) where HM is the machine/program, P is the program and I is the input. On taking input the both arguments the machine HM will tell that the program P either halts or not.
If we can design such a program this allows us to write another program we call this program CM(X) where X is any program(taken as argument) and according to the definition of the program CM(X) shown in the figure.

```
HM ( P,I)
{  Halt

or
May not Halt
}
```

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

}
```

In the program CM(X) we call the function HM(X), which we have already defined and to HM() we pass the arguments (X, X), according to the definition of HM() it can take two arguments i.e one is program and another is the input.Now in the second program we pass X as a program and X as input to the function HM().We know that the program HM() gives two output either "Halt" or "Not Halt".But in case second program, when HM(X, X) will halt loop body tells to go in loop and when it doesn't halt that means loop, it is asked to return.

Now we take one more situation where the program CM is passed to CM() function as an argument. Then there would be some impossibility, i.e., a condition arises which is not possible.

```
HM ( P,I)
{  Halt

or
May not Halt
}
```

```
CM ( X)
{
  if (HM( X,X)  ==HALT)
        loop forever;

  else
        return;
}
```

if we run CM on itself:

CM(CM,CM)

HM(CM,CM) ==HALT

```
{      Loop forever;
// it means it will never halt;
}
```

HM(CM,CM)==NOT HALT

```
{      Halt;
// it will never halt because of the
above non-halting condition;
}
```

It is impossible for outer function to halt if its code (inner body) is in loop and also it is impossible for outer non halting function to halt even after its inner code is halting. So the both condition is non halting for CM machine/program even we had assumed in the beginning that it would halt.So this is the contradiction and we can say that our assumption was wrong and this problem, i.e., halting problem is undecidable.

This is how we proved that halting problem is undecidable.