UNIT-5

Undecidable Problems:

A problem is undecidable if there is no Turing machine which will always halt in finite amount of time to give answer as 'yes' or 'no'. An undecidable problem has no algorithm to determine the answer for a given input.

Examples:

- Ambiguity of context-free languages: Given a context-free language, there is no Turing machine which will always halt in finite amount of time and give answer whether language is ambiguous or not.
- Equivalence of two context-free languages: Given two context-free languages, there is no Turing machine which will always halt in finite amount of time and give answer whether two context free languages are equal or not.
- Everything or completeness of CFG: Given a CFG and input alphabet, whether CFG will generate all possible strings of input alphabet ($\sum^*$) is undecidable.
- Regularity of CFL, CSL, REC and REC: Given a CFL, CSL, REC or REC, determining whether this language is regular is undecidable.

Decidable Problems:

A problem is decidable if we can construct a Turing machine which will halt in finite amount of time for every input and give answer as 'yes' or 'no'. A decidable problem has an algorithm to determine the answer for a given input.

Examples:

- Equivalence of two regular languages: Given two regular languages, there is an algorithm and Turing machine to decide whether two regular languages are equal or not.
- Finiteness of regular language: Given a regular language, there is an algorithm and Turing machine to decide whether regular language is finite or not.

- Emptiness of context free language: Given a context free language, there is an algorithm whether CFL is empty or not.

Rice Theorem:

Any nontrivial property about the language recognized by a Turing machine is undecidable.

A property about Turing machines can be represented as the language of all Turing machines, encoded as strings, that satisfy that property. The property P is about the language recognized by Turing machines if whenever L(M)=L(N) then P contains (the encoding of) M if it contains (the encoding of) N. The property is non-trivial if there is at least one Turing machine that has the property, and at least one that hasn't.

Proof: Without limitation of generality, we may assume that a Turing machine that recognizes the empty language does not have the property P. For if it does, just take the complement of P. The undecidability of that complement would immediately imply the undecidability of P.

In order to arrive at a contradiction, suppose P is decidable, i.e. there is a halting Turning machine B that recognizes the descriptions of Turing machines that satisfy P. Using B we can construct a Turning machine A that accepts the language {(M,w)| M is the description of a Turing machine that accepts the string w}. As the latter problem is undecidable this will show that B cannot exists and P must be undecidable as well.

Let MP be a Turing machine that satisfies P (as P is non-trivial there must be one). Now A operates as follows:

1. On input (M,w), create a (description of a) Turing machine C(M,w) as follows:
    a. On input x, let the Turing machine M run on the string w until it accepts (so if it doesn't accept C(M,w) will run forever).
    b. Next run MP on x. Accept iff MP does.
2. Note that C(M,w) accepts the same language as MP if M accepts w; C(M,w) accepts the empty language if M does not accept w.

Thus if M accepts w the Turing machine C(M,w) has the property P, and otherwise it doesn't.

3. Feed the description of C(M,w) to B. If B accepts, accept the input (M,w); if B rejects, reject

Post Correspondence Problem:

In this section, we will discuss the undecidability of string and not of Turing machines. The undecidability of the string is determined with the help of Post's Correspondence Problem (PCP). Let us define the PCP.

"The Post's correspondence problem consists of two lists of string that are of equal length over the input. The two lists are A = $w_1, w_2, w_3, \ldots, w_n$ and B = $x_1, x_2, x_3, \ldots x_n$ then there exists a non-empty set of integers $i_1, i_2, i_3, \ldots$, in such that,

$w_1, w_2, w_3, \ldots w_n = x_1, x_2, x_3, \ldots x_n$"

To solve the post correspondence problem, we try all the combinations of $i_1, i_2, i_3, \ldots$, in to find the w1 = x1 then we say that PCP has a solution.

Example 1:

Find whether the lists: M = (abb, aa, aaa) and N = (bba, aaa, aa) have a Post Correspondence Solution?

Solution:

	x_1	x_2	x_3
M	abb	aa	aaa
N	bba	aaa	aa

Here,

$x_2x_1x_3$ = ‘aaabbaaa’

and $y_2y_1y_3$ = ‘aaabbaaa’

We can see that

$x_2x_1x_3 = y_2y_1y_3$

Hence, the solution is i = 2, j = 1, and k = 3.

Example 2
Find whether the lists **M = (ab, bab, bbaaa)** and **N = (a, ba, bab)** have a Post Correspondence Solution?

Solution

	x_1	x_2	x_3
M	ab	bab	bbaaa
N	a	ba	bab

In this case, there is no solution because −

$| x_2x_1x_3 | \neq | y_2y_1y_3 |$ (Lengths are not same)

Hence, it can be said that this Post Correspondence Problem is undecidable.

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 go 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.

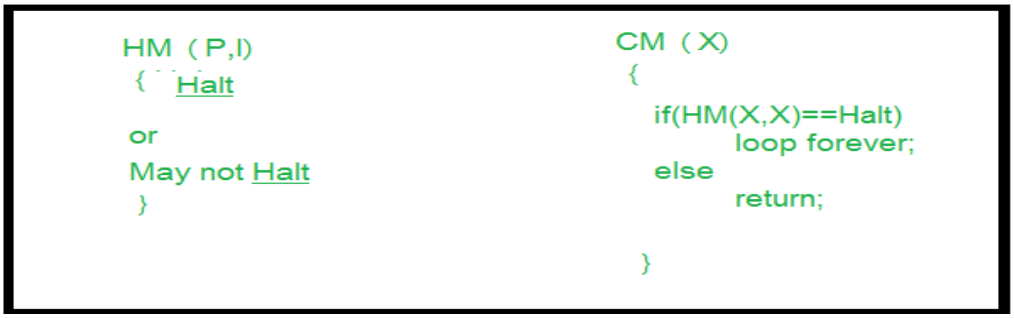
The answer is no we cannot design a generalized algorithm which can appropriately say that given a program will ever halt or not?
The only way is to run the program and check whether it halts or not.
We can refrain the halting problem question in such a way also: Given a program written in some programming 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 generalized 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?

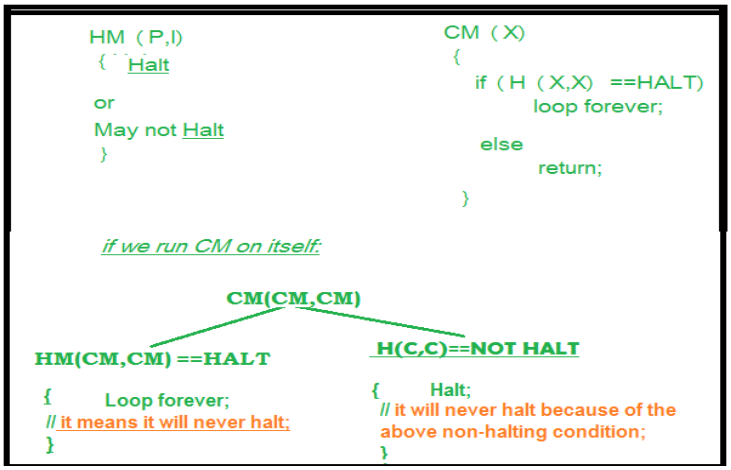
Solution: Let us assume that we can design that kind of machine called as HM(P, I) where HM is the machine/program, P is the program and I is the input. On taking input the both arguments the machine HM will tell that the program P either halts or not.

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.



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.



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.

Recursive and recursive enumerable language:

A language is recursive if there exists a Turing machine that accepts every string in the language and rejects if it is not in the language.

for example, let's take Turing machine M and String w: if string w is a member of the Turing machine M, then M halts in its final state otherwise it rejects the computation. ==>==> A language is recursive enumerable if there exists a Turing machine that accepts every string in the language and rejects if it is not in the language may be loop forever.

for example, let's take Turing machine M and String w: if string w is in the language, then M halts in its final state. Otherwise, it rejects the computation or may be run forever

Recursive languages are decidable by some Turing Machine, i.e., there is a TM that can, given any input string (over the appropriate alphabet) correctly answer yes if the string is in the language, or no if it isn't.

A language is recursive enumerable if there exists a TM that keeps outputting strings that belong to the language (and only such strings), such that eventually every string in the language will be in the output.

Recursively enumerable languages are only recognized, i.e., there exists a Turing Machine that accepts when the string is in the language but it may loop forever if the string is not in the language.

Recursively enumerable Turing machine if it not accepts a string may halt in non-final state or loop forever which is not the case for recursive languages

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= {anbncn|n>=1} is recursive because we can construct a turing machine which will move to final state if the string is of the form anbncn 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. For Example:

  L1= $\{a^nb^nc^n|n>=0\}$ L1*= $\{ a^nb^nc^n||n>=0\}^*$ is also recursive.

- **Intersection and complement**: If L1 and If L2 are two recursive languages, their intersection L1 ∩ L2 will also be recursive. For Example: L1= $\{a^nb^nc^ndm|n>=0$ and $m>=0\}$ L2= $\{a^nb^nc^nd^n|n>=0$ and $m>=0\}$ L3=L1 ∩ L2 = $\{ a^nb^nc^nd^n |n>=0\}$ will be recursive.

- L1 says n no. of a's followed by n no. of b's followed by n no. of c's and then any no. of d's. L2 says any no. of a's followed by n no. of b's followed by n no. of c's followed by n no. of d's. Their intersection says n no. of a's followed by n no. of b's followed by n no. of c's followed by n no. of d's. So it can be decided by turing machine, hence recursive.
  Similarly, complement of recursive language L1 which is ∑*-L1, will also be recursive.

Are all languages either recursive or recursively enumerable (decidable or recognizable)?

Take the language HALT(M,w) that consists of all Turing machines that halt for input w.

Turing proved that this language is not decidable. It is, however, recursively enumerable because we can test all Turing machines with input w and print them if they ever halt.

If a language and its complement are recursively enumerable, it is decidable. Why? Simply execute the enumerators for the language and its complement with and wait until one of them outputs your input. If it is the enumerator of the language, your input is in the language. If it is the other enumerator, your input is not in the language.

It follows directly that the complement of HALT(M,w) cannot be recursively enumerable because else, HALT(M,w) would be decidable.

Computational Complexity Theory:

Computational complexity theory is a subfield of theoretical computer science one of whose primary goals is to classify and compare the practical difficulty of solving problems about finite combinatorial objects – e.g., given two natural numbers n and m, are they relatively prime? Given a propositional formula ϕ, does it have a satisfying assignment? If we were to play chess on a board of size $n \times n$, does white have a winning strategy from a given initial position? These problems are equally difficult from the standpoint of classical computability theory in the sense that they are all effectively decidable. Yet they still appear to differ significantly in practical difficulty. For having been supplied with a pair of numbers $m>n>0$, it is possible to determine their relative primality by a method (Euclid's algorithm) which requires a number of steps proportional to $\log(n)$. On the other hand, all known methods for solving the latter two problems require a 'brute force' search through a large class of cases which increase at least exponentially in the size of the problem instance.

Complexity theory attempts to make such distinctions precise by proposing a formal criterion for what it means for a mathematical problem to be feasibly decidable – i.e. that it can be solved by a conventional Turing machine in a number of steps which is proportional to a polynomial function of the size of its input. The class of problems with this property is known as P – or polynomial time – and includes the first of the three problems described above. P can be formally shown to be distinct from certain other classes such as EXP – or exponential time – which includes the third problem from above. The second problem from above belongs to a complexity class known as NP – or non-deterministic polynomial time – consisting of those problems which can be correctly decided by some computation of a non-deterministic Turing machine in a number of steps which is a polynomial function of the size of its input. A famous conjecture – often regarded as the most fundamental in all of theoretical computer science – states that P is also properly contained in NP – i.e. P NP.

Big-Oh notation : As mentioned above, we will typically measure the computational e ciency algorithm as the number of a basic operations it performs as a function of its input length. That is, the e ciency of an algorithm can be captured by a function T from the set of natural numbers N to itself such that T(n) is equal to the maximum number of basic operations that the algorithm performs on inputs of length n. However, this function is sometimes be overly dependent on the low-level details of our definition of a basic operation. For example, the addition algorithm will take about three times more operations if it uses addition of

single digit binary (i.e., base 2) numbers as a basic operation, as opposed to decimal (i.e., base 10) numbers.

The basic question that computational complexity theory tries to answer is: Given a problem X, and a machine model M, how long does it take to solve X using a machine from M? Unfortunately, we can only rarely answer this question. So, the real questions of complexity theory are:

1. How can we classify problems into complexity classes based on their apparent di culty?

2. What relationships can we established between these complexity classes?

3. What techniques might we be able to use to resolve relationships between complexity classes whose status is still open?

The most famous of these questions center around the P vs. NP problem. Here P consists of problems that can be solved in time polynomial in the size of the input on reasonable computational devices, NP consists of problems whose solutions can be verified in time polynomial in the size of the input, and the big question is whether there are any problems in NP that are not also in P.

Computational complexity theory also includes questions about other computational resources. In addition to time, we can ask about how much space it takes to solve a particular problem. The space class analogous to P is L, the class of problems that can be solved using space logarithmic in the size of the input. Because a log-space machine can only have polynomial many distinct states, any problem solvable in L is also solvable in P, and indeed L is the largest space complexity class that includes only problems we can expect to solve e ciently in practice. As with P, a big open question is whether solving problems in log space is any harder than checking the solutions. In other words, is L equal to its nondeterministic counterpart NL? Other classes of interest include PSPACE, the class of problems solvable using polynomial space, which contains within it an entire hierarchy of classes that are to NP what NP is to P.

Complexity:

The time complexity of an execution is the number of steps until the machine halts. Typically, we will try to bound the time complexity as a function of the size n of the input, defined as the number of cells occupied by the input, excluding the infinite number of blanks that surround it.

The space complexity is the number of tape cells used by the computation. We have to be a little careful to define what it means to use a cell. A naive approach is just to count the number of cells that hold a non-blank symbol at any step of the computation, but this allows cheating (on a multi-tape Turing machine) because we can simulate an unbounded counter by writing a single non-blank symbol to one of our work tapes and using the position of the head relative to this symbol as the counter value.

Asymptotic notation In computing time and space complexities, we want to ignore constant factors and performance on small inputs, because we know that constant factors depend strongly on features of our model that we usually don't care much about, and performance on small inputs can always be faked by baking a large lookup table into our finite-state controller. As in the usual analysis of algorithms, we get around these issues by expressing performance using asymptotic notation. This section gives a brief review of asymptotic notation as used in algorithm analysis and computational complexity theory. Given two non-negative functions f(n) and g(n), we say that:

• $f(n)= O(g(n))$ if there exist constants $c > 0$ and N such that $f(n)\leq c\cdot g(n)$ for all $n \geq N$.

• $f(n)=\Omega(g(n))$ if there exist constants $c > 0$ and N such that $f(n)\geq c\cdot(gn)$ for all $n \geq N$

• $f(n)=\Theta(g(n))$ if $f(n)= O(g(n))$ and $f(n)=\Omega(g(n))$, or equivalently if there exist constants $c1 > 0$, $c2 > 0$, and N such that $c1 \cdot g(n) \leq f(n)\leq c2\cdot g(n)$ for all $n \geq N$.

• $f(n) = o(g(n))$ if for any constant $c > 0$, there exists a constant N such that $f(n)\leq c\cdot g(n)$ for all $n \geq N$.

• $f(n) = \omega(g(n))$ if for any constant $c > 0$, there exists a constant N such that $f(n)\geq c\cdot g(n)$ for all $n \geq N$.

P, NP, NP-Complete and NP-Hard Problems:

P- Polynomial time solving. Problems which can be solved in polynomial time, which take time like O(n), O(n2), O(n3). Eg: finding maximum element in an array or to check whether a string is palindrome or not. so, there are many problems which can be solved in polynomial time.

P is a complexity class that represents the set of all decision problems that can be solved in polynomial time.

That is, given an instance of the problem, the answer yes or no can be decided in polynomial time.

Example

Given a connected graph G, can its vertices be colored using two colors so that no edge is monochromatic?

Algorithm: start with an arbitrary vertex, color it red and all of its neighbors blue and continue. Stop when you run out of vertices or you are forced to make an edge have both of its endpoints be the same color.

NP- Non deterministic Polynomial time solving. Problem which can't be solved in polynomial time like TSP (travelling salesman problem) or an easy example of this is subset sum: given a set of numbers, does there exist a subset whose sum is zero?

but NP problems are checkable in polynomial time means that given a solution of a problem, we can check that whether the solution is correct or not in polynomial time.

Some think NP as non-Polynomial. But actually, it is Non-deterministic Polynomial time. i.e.; "yes/no" instances of these problems can be solved in polynomial time by a non-deterministic Turing machine and hence can take up to exponential time (some problems can be solved in sub-exponential but super polynomial time) by a deterministic Turing machine. In other words, these problems can be verified (if a solution is given, say if it is correct or wrong) in polynomial time by a deterministic Turing machine (or equivalently our computer). Examples include all P problems. One example of a problem not in P but in NP is Integer Factorization.

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

Intuitively, these are the problems that are at least as hard as the NP-complete problems. Note that NP-hard problems do not have to be in NP, and they do not have to be decision problems.

The precise definition here is that a problem `X` is NP-hard, if there is an NP-complete problem `Y`, such that `Y` is reducible to `X` in polynomial time.

But since any NP-complete problem can be reduced to any other NP-complete problem in polynomial time, all NP-complete problems can be reduced to any NP-hard problem in polynomial time. Then, if there is a solution to one NP-hard problem in polynomial time, there is a solution to all NP problems in polynomial time.

Example

The halting problem is an NP-hard problem. This is the problem that given a program `P` and input `I`, will it halt? This is a decision problem but it is not in NP. It is clear that any NP-complete problem can be reduced to this one. As another example, any NP-complete problem is NP-hard.

NP-Hard Problems

The following problems are NP-Hard

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

Difference Between NP-Hard and NP-Complete Problem:

Parameters	NP-Hard Problem	NP-Complete Problem
Meaning and Definition	One can only solve an NP-Hard Problem X only if an NP-Complete Problem Y exists. It then becomes reducible to problem X in a polynomial time.	Any given problem X acts as NP-Complete when there exists an NP problem Y- so that the problem Y gets reducible to the problem X in a polynomial line.
Presence in NP	The NP-Hard Problem does not have to exist in the NP for anyone to solve it.	For solving an NP-Complete Problem, the given problem must exist in both NP-Hard and NP Problems.
Decision Problem	This type of problem need not be a Decision problem.	This type of problem is always a Decision problem (exclusively).
Example	Circuit-satisfactory, Vertex cover, Halting problems, etc., are a few examples of NP-Hard Problems.	A few examples of NP-Complete Problems are the determination of the Hamiltonian cycle in a graph, the determination of the satisfaction level of a Boolean formula, etc.