Time and Space Complexity of a Turing Machine

For a Turing machine, the time complexity refers to the measure of the number of times the tape moves when the machine is initialized for some input symbols and the space complexity is the number of cells of the tape written.

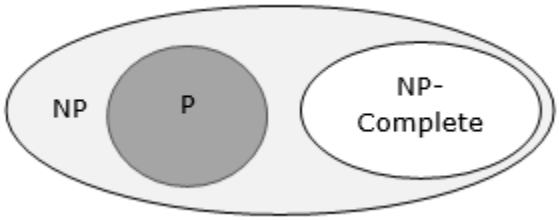
Time complexity all reasonable functions –

T(n) = O(n log n)

TM's space complexity –

S(n) = O(n)

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

In this context, now we will discuss TSP is NP-Complete

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

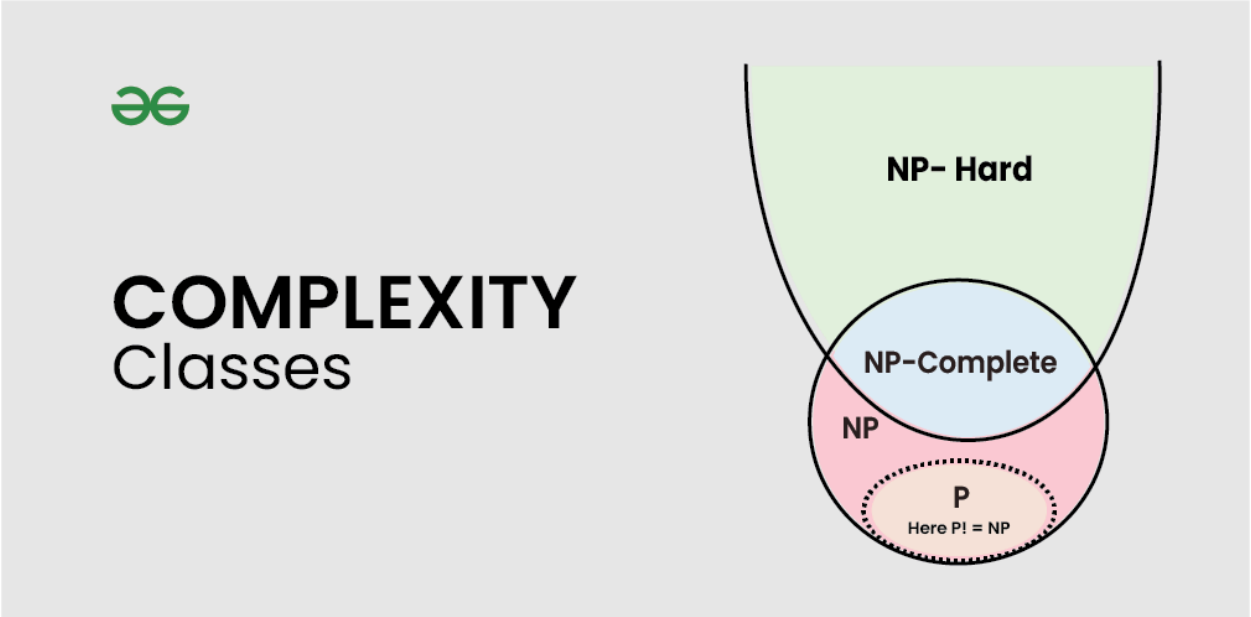
To prove **TSP is NP-Complete**, first we have to prove that **TSP belongs to NP**. In TSP, we find a tour and check that the tour contains each vertex once. Then the total cost of the edges of the tour is calculated. Finally, we check if the cost is minimum. This can be completed in polynomial time.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

In computer science, there exist some problems whose solutions are not yet found, the problems are divided into classes known as **Complexity Classes**. In complexity theory, a Complexity Class is a set of problems with related complexity. These classes help scientists to group problems based on how much time and space they require to solve problems and verify the solutions. It is the branch of the theory of computation that deals with the resources required to solve a problem.

The common resources are time and space, meaning how much time the algorithm takes to solve a problem and the corresponding memory usage.

- The time complexity of an algorithm is used to describe the number of steps required to solve a problem, but it can also be used to describe how long it takes to verify the answer.
- The space complexity of an algorithm describes how much memory is required for the algorithm to operate.

Complexity classes are useful in organising similar types of problems.



Types of Complexity Classes

This article discusses the following complexity classes:

1. **P Class**
2. **NP Class**
3. **CoNP Class**
4. **NP-hard**
5. **NP-complete**

P Class

The P in the P class stands for **Polynomial Time.** It is the collection of decision problems(problems with a “yes” or “no” answer) that can be solved by a deterministic machine in polynomial time.

Features:

- The solution to **P problem**s is easy to find.

- **P** is often a class of computational problems that are solvable and tractable. Tractable means that the problems can be solved in theory as well as in practice. But the problems that can be solved in theory but not in practice are known as intractable.

This class contains many problems:

1. **Calculating the greatest common divisor.**
2. **Finding a maximum matching.**
3. **Merge Sort**

NP Class

The NP in NP class stands for **Non-deterministic Polynomial Time**. It is the collection of decision problems that can be solved by a non-deterministic machine in polynomial time.

Features:

- The solutions of the NP class are hard to find since they are being solved by a non-deterministic machine but the solutions are easy to verify.
- Problems of NP can be verified by a Turing machine in polynomial time.

Example:

Let us consider an example to better understand the **NP class**. Suppose there is a company having a total of **1000** employees having unique employee **IDs**.

Assume that there are **200** rooms available for them. A selection of **200** employees must be paired together, but the CEO of the company has the data of some employees who can't work in the same room due to personal reasons.
This is an example of an **NP** problem. Since it is easy to check if the given choice of **200** employees proposed by a coworker is satisfactory or not i.e. no pair taken from the coworker list appears on the list given by the CEO. But generating such a list from scratch seems to be so hard as to be completely impractical.

It indicates that if someone can provide us with the solution to the problem, we can find the correct and incorrect pair in polynomial time. Thus for the **NP** class problem, the answer is possible, which can be calculated in polynomial time.

This class contains many problems that one would like to be able to solve effectively:

1. **Boolean Satisfiability Problem (SAT).**
2. **Hamiltonian Path Problem.**
3. **Graph coloring.**

Co-NP Class

Co-NP stands for the complement of NP Class. It means if the answer to a problem in Co-NP is No, then there is proof that can be checked in polynomial time.

Features:

- If a problem X is in NP, then its complement X' is also in CoNP.

- For an NP and CoNP problem, there is no need to verify all the answers at once in polynomial time, there is a need to verify only one particular answer “yes” or “no” in polynomial time for a problem to be in NP or CoNP.

Some example problems for CoNP are:

1. **To check prime number.**
2. **Integer Factorization.**

NP-hard class

An NP-hard problem is at least as hard as the hardest problem in NP and it is a class of problems such that every problem in NP reduces to NP-hard.

Features:

- All NP-hard problems are not in NP.
- It takes a long time to check them. This means if a solution for an NP-hard problem is given then it takes a long time to check whether it is right or not.
- A problem A is in NP-hard if, for every problem L in NP, there exists a polynomial-time reduction from L to A.

Some of the examples of problems in Np-hard are:

1. **Halting problem.**
2. **Qualified Boolean formulas.**
3. **No Hamiltonian cycle.**

NP-complete class

A problem is NP-complete if it is both NP and NP-hard. NP-complete problems are the hard problems in NP.

Features:

- NP-complete problems are special as any problem in NP class can be transformed or reduced into NP-complete problems in polynomial time.
- If one could solve an NP-complete problem in polynomial time, then one could also solve any NP problem in polynomial time.

Some example problems include:

1. **Hamiltonian Cycle.**
2. **Satisfiability.**
3. **Vertex cover.**

Complexity Class	Characteristic feature
P	Easily solvable in polynomial time.

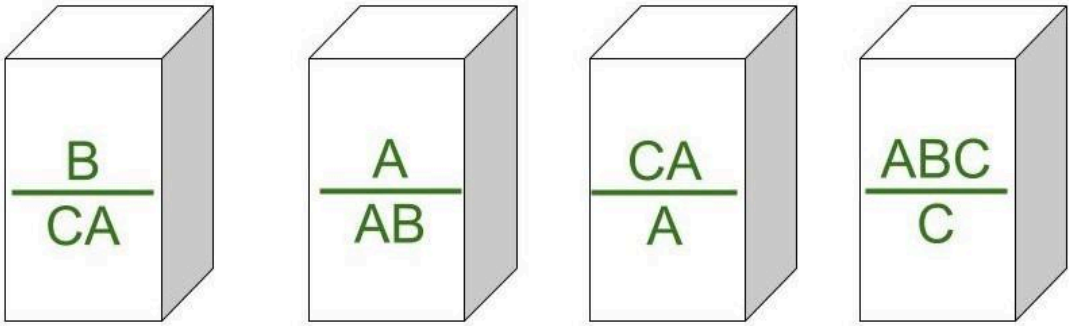
NP	Yes, answers can be checked in polynomial time.
Co-NP	No, answers can be checked in polynomial time.
NP-hard	All NP-hard problems are not in NP and it takes a long time to check them.
NP-complete	A problem that is NP and NP-hard is NP-complete.

Post Correspondence Problem is a popular undecidable problem that was introduced by Emil Leon Post in 1946. It is simpler than Halting Problem. In this problem we have N number of **Dominos** (tiles). The aim is to arrange tiles in such order that string made by Numerators is same as string made by Denominators. In simple words, lets assume we have two lists both containing N words, aim is to find out concatenation of these words in some sequence such that both lists yield same result. Let's try understanding this by taking two **lists A** and **B**

```
A=[aa, bb, abb] and B=[aab, ba, b]
```

Now for sequence 1, 2, 1, 3 first list will yield aabbaaabb and second list will yield same string aabbaaabb. So the solution to this PCP becomes 1, 2, 1, 3. Post Correspondence Problems can be represented in two ways:

1. Domino's Form :



2. Table Form :

	Numerator	Denominator
1	B	CA
2	A	AB
3	CA	A
4	ABC	C

Lets consider following examples. **Example-1:**

	A	B
1	1	111
2	10111	10
3	10	0

Explanation –

- **Step-1:** We will start with tile in which numerator and denominator are starting with same number, so we can start with either 1 or 2. Lets go with **second** tile, string made by numerator- 10111, string made by denominator is 10.
- **Step-2:** We need 1s in denominator to match 1s in numerator so we will go with **first** tile, string made by numerator is 10111 1, string made by denominator is 10 111.
- **Step-3:** There is extra 1 in numerator to match this 1 we will add **first** tile in sequence, string made by numerator is now 10111 1 1, string made by denominator is 10 111 111.

- **Step-4:** Now there is extra 1 in denominator to match it we will add **third** tile, string made by numerator is 10111 1 1 10, string made by denominator is 10 111 111 0.

```
Final Solution - 2 1 1 3
String made by numerators: 101111110
String made by denominators: 101111110
```

- As you can see, strings are same.

Example-2:

	A	B
1	100	1
2	0	100
3	1	00

Explanation –

- **Step-1:** We will start from tile **1** as it is our only option, string made by numerator is 100, string made by denominator is 1.

- **Step-2:** We have extra 00 in numerator, to balance this only way is to add tile **3** to sequence, string made by numerator is 100 1, string made by denominator is 1 00.
- **Step-3:** There is extra 1 in numerator to balance we can either add tile **1** or tile **2**. Lets try adding tile 1 first, string made by numerator is 100 1 100, string made by denominator is 1 00 1.
- **Step-4:** There is extra 100 in numerator, to balance this we can add **1st tile** again, string made by numerator is 100 1 100 100, string made by denominator is 1 00 1 1 1. The 6th digit in numerator string is 0 which is different from 6th digit in string made by denominator which is 1.

To understand better the halting problem, we must know

Decidability

,

Undecidability

and

Turing machine

,

decision problems

and also a theory named as Computability theory and Computational complexity theory. Some important terms:

- **Computability theory** – The branch of theory of computation that studies which problems are computationally solvable using different model. In computer science, the computational complexity, or simply complexity of an algorithm is the amount of resources required for running it.
- **Decision problems** – A decision problem has only two possible outputs (yes or no) on any input. In computability theory and computational complexity theory, a decision problem is a problem that can be posed as a yes-no question of the input values. Like is there any solution to a particular problem? The answer would be either a yes or no. A decision problem is any arbitrary yes/no question on an infinite set of inputs.
- **Turing machine** – A Turing machine is a mathematical model of computation. A Turing machine is a general example of a CPU that controls all data manipulation done by a computer. Turing machine can be halting as well as non halting and it depends on algorithm and input associated with the algorithm.

Now, lets discuss Halting problem:

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.

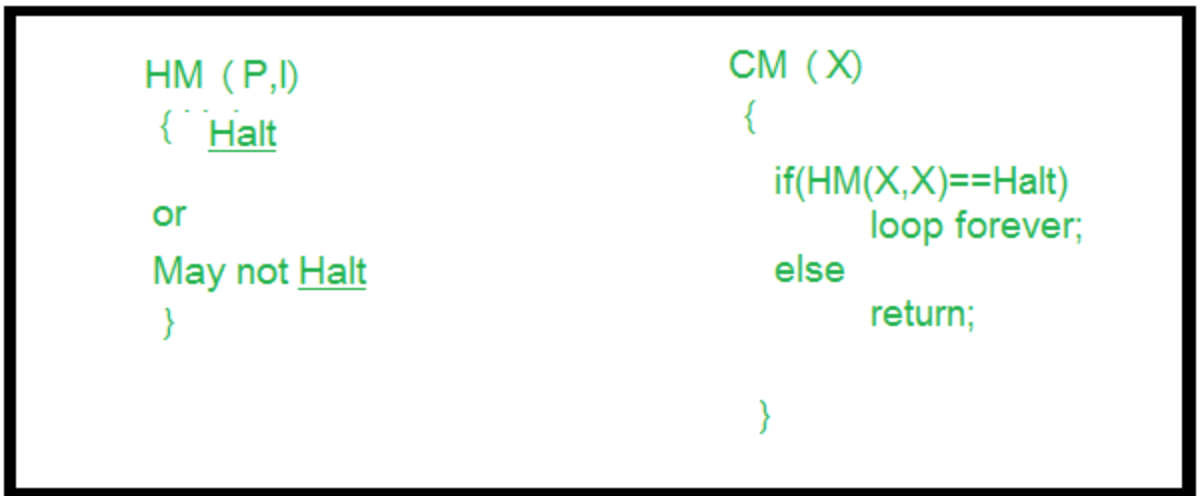
Proof by Contradiction –

Problem statement:

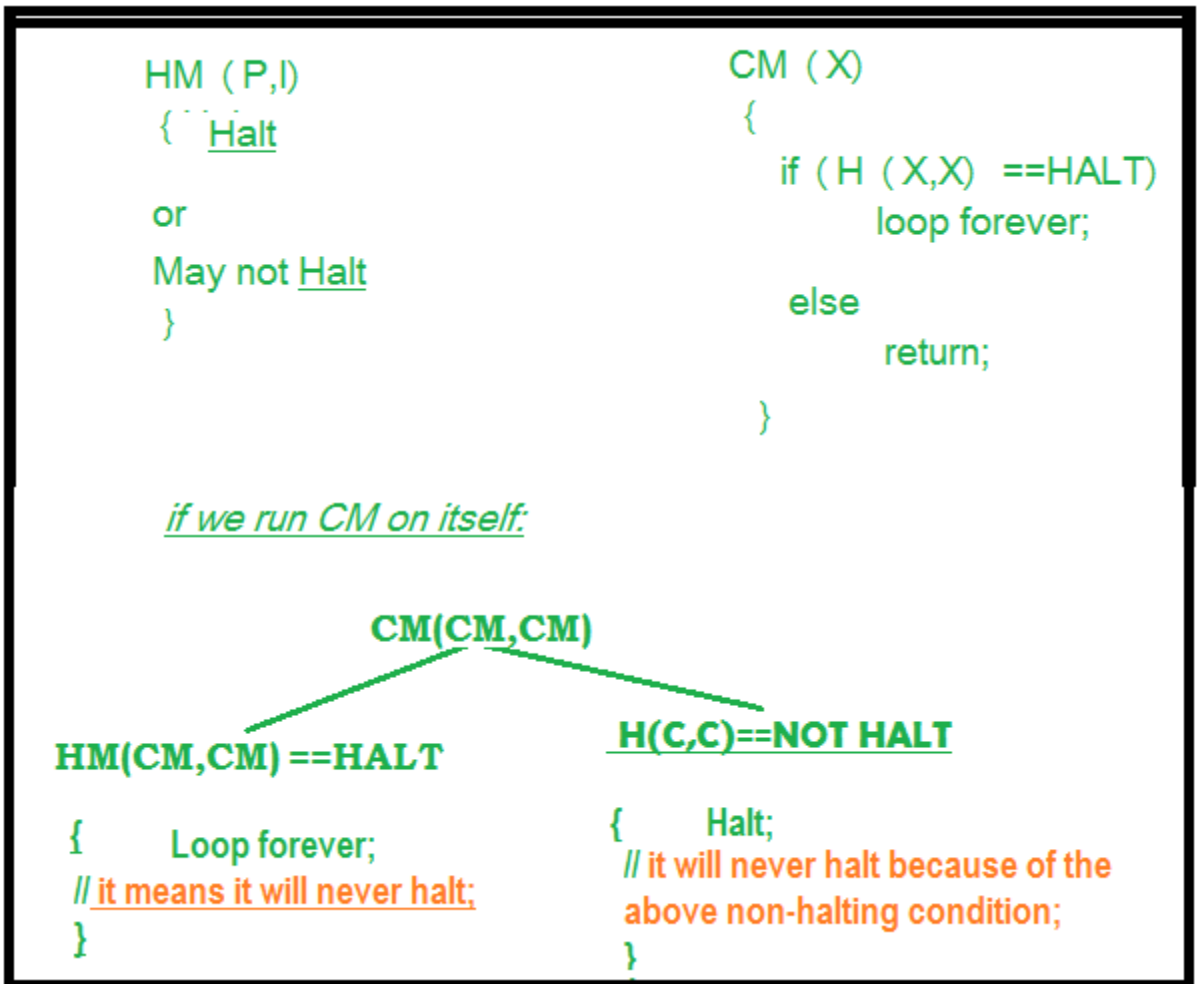
Can we design a machine which if given a program can find out if that program will always halt or not halt on a particular input?

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.

In the Theory of Computation, problems can be classified into decidable and undecidable categories based on whether they can be solved using an algorithm. A **decidable problem** is one for which a solution can be found in a finite amount of time, meaning there exists an algorithm that can always provide a correct answer. While an **undecidable problem** is one where no

algorithm can be constructed to solve the problem for all possible inputs. In this article, we will discuss Decidable and Undecidable problems in detail.

What are Decidable Problems?

A problem is said to be **Decidable** if we can always construct a corresponding **algorithm** that can answer the problem correctly. We can intuitively understand Decidable issues by considering a simple example. Suppose we are asked to compute all the prime numbers in the range of 1000 to 2000. To find the **solution** to this problem, we can easily construct an algorithm that can enumerate all the prime numbers in this range.

Now talking about Decidability in terms of a Turing machine, a problem is said to be a Decidable problem if there exists a corresponding Turing machine that **halts** on every input with an answer- **yes or no**. It is also important to know that these problems are termed **Turing Decidable** since a Turing machine always halts on every input, accepting or rejecting it.

Semi Decidable Problems

Semi-decidable problems are those for which a Turing machine halts on the input accepted by it but it can either halt or loop forever on the input which the Turing Machine rejects. Such problems are termed as **Turing Recognisable** problems.

Example

We will now consider some few important **Decidable** problems:

- **Are two regular languages L and M equivalent:** We can easily check this by using Set Difference operation. L-M =Null and M-L =Null. Hence (L-M) U (M-L) = Null, then L,M are equivalent.

- **Membership of a CFL:** We can always find whether a string exists in a given CFL by using an algorithm based on dynamic programming.
- Emptiness of a CFL By checking the production rules of the CFL we can easily state whether the language generates any strings or not.

What are Undecidable Problems?

The problems for which we can't construct an algorithm that can answer the problem correctly in finite time are termed as Undecidable Problems. These problems may be partially decidable but they will never be decidable. That is there will always be a condition that will lead the Turing Machine into an infinite loop without providing an answer at all.

We can understand Undecidable Problems intuitively by considering **Fermat's Theorem**, a popular Undecidable Problem which states that no three positive integers a, b and c for any n>2 can ever satisfy the equation: a^n + b^n = c^n. If we feed this problem to a Turing machine to find such a solution which gives a contradiction then a Turing Machine might run forever, to find the suitable values of n, a, b and c. But we are always unsure whether a contradiction exists or not and hence we term this problem as an Undecidable Problem.

Example

These are few important **Undecidable Problems**:

- **Whether a CFG generates all the strings or not:** As a Context Free Grammar (CFG) generates infinite strings, we can't ever reach up to the last string and hence it is Undecidable.

- **Whether two CFG L and M equal:** Since we cannot determine all the strings of any CFG, we can predict that two CFG are equal or not.
- **Ambiguity of CFG:** There exist no algorithm which can check whether for the ambiguity of a Context Free Language (CFL). We can only check if any particular string of the CFL generates two different parse trees then the CFL is ambiguous.
- **Is it possible to convert a given ambiguous CFG into corresponding non-ambiguous CFL:** It is also an Undecidable Problem as there doesn't exist any algorithm for the conversion of an ambiguous CFL to non-ambiguous CFL.
- **Is a language Learning which is a CFL, regular:** This is an Undecidable Problem as we can not find from the production rules of the CFL whether it is regular or not.

*The most well-known undecidable problem is the **Halting Problem**, which asks whether a given program will stop running (halt) or continue running forever for a particular input. It has been proven that no algorithm can solve this problem for all programs and inputs.*

Some more **Undecidable Problems** related to Turing machine:

- **Membership** problem of a Turing Machine
- **Finiteness** of a Turing Machine
- **Emptiness** of a Turing Machine
- Whether the language accepted by Turing Machine is regular or CFL

Difference Between Decidable and Undecidable Problems

Aspect	Decidable Problems	Undecidable Problems
Definition	Problems that can be solved by an algorithm that always gives a correct answer in a finite time.	Problems where no algorithm can give a solution for all possible cases.
Solvability	Always solvable using a step-by-step process (algorithm).	Cannot be solved for all inputs using a single algorithm.

Algorithm	There is an algorithm that works for every input and always finishes with an answer.	No algorithm can solve the problem for every input.
Halting	The algorithm stops (halts) and gives an answer for every input.	The algorithm might never stop for some inputs, or no algorithm exists.
Examples	Problems like checking if a number is even or odd, or if a string belongs to a regular language (like finding a match in a search).	Examples include the Halting Problem, where you can't always tell if a program will finish running or run forever.
Decision Procedure	There's a clear method to always reach a correct conclusion.	No guaranteed method exists to solve the problem in every case.

Complexity	May be complex but can always be computed.	Too complex to compute in general, and no universal solution exists.
Applications	Useful in practical computing tasks like compiling code or searching for text patterns.	Helps understand the limits of what computers can do, showing what problems are beyond computation.

Conclusion

In conclusion, decidable and undecidable problems highlight the boundaries of what computers can and cannot solve. **Decidable problems** have solutions that can always be found by an algorithm, making them predictable and useful in everyday computing tasks. On the other hand, **undecidable problems** demonstrate the limits of computation, where no algorithm can provide a solution for every possible case.

Understanding the difference between these two types of problems helps us recognize which problems are solvable and which ones are beyond the reach of algorithms.

Decidable and Undecidable Problems in Theory of Computation – FAQs

Is it possible for a problem to be partially decidable?

*Yes, a problem is called **partially decidable** (or **semi-decidable**) if an algorithm can provide a solution for some inputs but may not halt or give an answer for others. In other words, it may solve some cases but not all.*

Are all NP problems undecidable?

No, NP problems (like the traveling salesman problem) are not undecidable. NP problems are difficult to solve quickly, but there exists an algorithm that can verify a solution efficiently if given one.

What is the role of Turing machines in deciding problems?

*A **Turing machine** is a theoretical model used to define what problems are decidable or undecidable. If a Turing machine can solve a problem for all inputs, it is decidable. If no Turing machine can solve the problem for all*

inputs, it is undecidable. Turing machines help establish the theoretical limits of computation.