**Computational Complexity**:

The complexity of computational problems can be discussed by choosing a specific abstract machine as a model of computation and considering how much time and/or space machine of that type require for the solution of that problem.

♦ A given problem can be solved by using more than one computational model i.e. there may be more than one TM that solve the problem. It is thus necessary to measure the qualities of alternative model to solve the same computational problem.

♦ The quality of a computational model is measured usually in terms of the resources needed by the algorithm for its execution.

♦ The two important resources used for executing a given algorithm are (i) Time (ii) memory, required to execute that algorithm.

♦ When estimating execution time (Time complexity) we are interested in growth rate and not in absolute time.

♦ Similarly, we are interested in growth rate of memory need (space complexity) rather than the absolute value of space.

♦ So the boundary time and boundary space for executing an algorithm are usually expressed in terms of known mathematical functions.

**Time and Space Complexity of a Turing Machine:**

The model of computation we have chosen is the Turing Machine. When a Turing machine answers a specific instance of a decision problem we can measure time as number of moves and the space as number of tape squares, required by the computation. The most obvious measure of the size of any instance is the length of input string. The worst case is considered as the maximum time or space that might be required by any string of that length.

The time and space complexity of a TM can be defined as;

Let T be a TM. The time complexity of T is the function Tt defined on the natural numbers as; for n ε N, Tt(n) is the maximum number of moves T can make on any input string of length n. If there is an input string x such that for |x|=n, T loops forever on input T, Tt(n) is undefined.

The space complexity of T is the function St defined as St (n) is the maximum number of the tape squares used by T for any input string of length n. If T is multi-tape TM, number of tape squares means maximum of the number of individual tapes. If for some input of length n, it causes T to loop forever, St (n) is undefined.

An algorithm for which the complexity measures St(n) increases with n, no more rapidly than a polynomial in n is said to be *polynomially bounded*; one in which it grows exponentially is said to be *exponentially bounded*.

**Intractability:**

The problems that can be solved by any computational model, probably TM, using no more time than some slowly growing function size of the input are called “tractable”, i.e. those problems solvable within reasonable time and space constraints (polynomial time). The problems that cannot be solved in polynomial time but requires super polynomial (exponential) time algorithm are called intractable or hard problems. There are many problems for which no algorithm with running time better than exponential time is known some of them are, traveling salesman problem, Hamiltonian cycles, and circuit satisfiability, etc.

To introduce intractability theory, the class P and class NP of problems solvable in polynomial time by deterministic and non-deterministic TM’s are essential. A solvable problem is one that can be solved by particular algorithm i.e. there is an algorithm to solve this problem. But in practice, the algorithm may require a lot of space and time. When the space and time required for implementing the steps of the particular algorithm are (polynomial) reasonable, we can say that the problem is tractable. Problems are intractable if the time required for any of the algorithm is at least f(n), where f is an exponential function of n.

**Complexity Classes:**

**Class P:**

The class P is the set of problems that can be solved by deterministic TM in polynomial time.

A language L is in class P if there is some polynomial time complexity T(n) such that L=L(M), for some Deterministic Turing Machine M of time complexity T(n).

**Class NP:**

The class NP is the set of problems that can be solved by a non-deterministic TM in polynomial time.

Formally, we can say a language L is in the class NP if there is a non-deterministic TM, M, and a polynomial time complexity T(n), such that L= L(M), and

when M is given an input of length n, there are no sequences of more than T(n) moves of M.

**Problems:**

**Abstract Problems:**

An abstract problem Q is a binary relation on a set I of problem instances and a set S of problem solutions. An abstract decision problem can be viewed as a function that maps the instance set I to the solution set {0,1}.

For example, if I = (G= (V, E), s, t, k) is an instance of the decision problem PATH, then PATH(I) = 1 if a shortest path from s to t has at most k edges and PATH(I) = 0 otherwise.

**Decision Problems:**

Decision problem D is a problem that has an answer as either “true” or “yes”, “1” or “false”, “ no” or “0”. For e.g. if we have the abstract shortest path with instances of the problem and the solution set as {0,1}, then we can transform that abstract problem by reformulating the problem as “Is there a path from u to v with at most k edges”. In this situation the answer is either yes or no.

**Optimization Problems:**

We encounter many problems where there are many feasible solutions and our aim is to find the feasible solution with the best value. This kind of problem is called optimization problem. For e.g. given the graph G, and the vertices u and v find the shortest path from u to v with minimum number of edges. The NP completeness does not directly deal with optimizations problems; however, we can translate the optimization problem to the decision problem.

**Reducibility:**

Reducibility is a way of converging one problem into another in such a way that, a solution to the second problem can be used to solve the first one.

Many complexity classes are defined using the concept of a reduction. A reduction is a transformation of one problem into another problem. It captures the informal notion of a problem being at least as difficult as another problem. For instance, if a problem X can be solved using an algorithm for Y, X is no more difficult than Y, and we say that X reduces to Y.

The most commonly used reduction is a polynomial-time reduction. This means that the reduction process takes polynomial time.

If we can transform instances of one problem in polynomial time into instances of a second problem that has the same answer –yes or –no then we say the first problem is polynomial time reducible to the second.

Eg: quadratic equation solver.

1. ax2+bx+c=0

try to solve

1. 3x+5=0 using 1.

We have reduced problem of linear equation to the problem of quadratic equation.

It can be reduced by transforming

Eg: a=0, b=3, c=5 from 2.

While transforming, transformation steps should be easy.

**NP- complete problems**

We say L is NP-complete if the following statements are true about L:

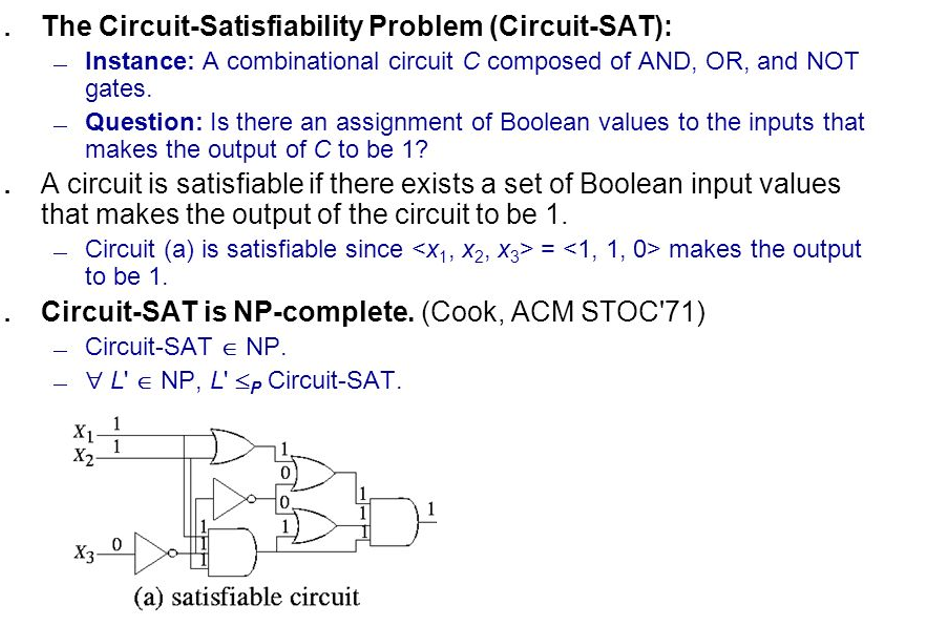
1. L is in NP
2. For every language L’ in NP there is a polynomial time reduction of L’ to L

**NP- hard problem:**

Some problems are so hard that although we can prove condition 2, we cannot prove condition 1

If so we call such problem a np-hard.

**Circuit Satisfiability:**



**Cook’s Theorem**

**Lemma: SAT is NP-hard**

**Proof: (This is not actual proof as given by cook, this is just a sketch)**

Take a problem V ε NP, let A be the algorithm that verifies V in polynomial time (this must be true since V ε NP). We can program A on a computer and therefore there exists a (huge) logical circuit whose input wires correspond to bits of the inputs x and y of A and which outputs 1 precisely when A(x, y) returns yes.

For any instance x of V let Ax be the circuit obtained from A by setting the x-input wire values according to the specific string x. The construction of Ax from x is our reduction function. If x is a yes instance of V, then the certificate y for x gives satisfying assignments for Ax. Conversely, if Ax outputs 1 for some assignments to its input wires, that assignment translates into a certificate for x.

**Theorem: SAT is NP-complete**

**Proof:**

To show that SAT is NP-complete we have to show two properties as given by the definition of NP-complete problems. The first property i.e. SAT is in NP

Circuit satisfiability problem (SAT) is the question “Given a Boolean combinational circuit, is it satisfiable? i.e. does the circuit has assignment sequence of truth values that produces the output of the circuit as 1?” Given the circuit satisfiability problem take a circuit x and a certificate y with the set of values that produce output 1, we can verify that whether the given certificate satisfies the circuit in polynomial time. So we can say that circuit satisfiability problem is NP.

This claims that SAT is NP. Now it is sufficient to show the second property holds for SAT. The proof for the second property i.e. SAT is NP-hard is from above lemma. This completes the proof.

**Undecidability:**

In computability theory, an undecidable problem is a decision problem for which it is impossible to construct a single algorithm that always leads to a correct “yes” or “no” answer- the problem is not decidable.

An **undecidable problem** consists of a family of instances for which a particular yes/no answer is required, such that there is no computer program that, given any problem instance as input, terminates and outputs the required answer after a finite number of steps. More formally, an undecidable problem is a problem whose language is not a recursive set or computable or decidable.

In computability theory, **the halting problem** is a decision problem which can be stated as follows:

*Given a description of a program and a finite input, decide whether the program finishes running or will run forever.*

Alan Turing proved in 1936 that a general algorithm running on a Turing machine that solves the halting problem for *all* possible program-input pairs necessarily cannot exist. Hence, the halting problem is *undecidable* for Turing machines.

**Halting Problem:**

**“Given a Turing Machine M and an input w, do M halts on w?”**

Algorithms may contain loops which may be infinite or finite in length. The amount of work done in an algorithm usually depends on data input. Algorithms may consist of various numbers of loops nested or in sequence. Thus, the halting problem asks the question; *“Given a program and an input to the program, determine if the program will eventually stop when it is given that input.”* The question is simply whether the given program will ever halt on a particular input.

**Trial Solution:** Just run the program with the given input. If the program stops, we know the program halts. But if the program does not stop in reasonable amount of time, we cannot conclude that it won’t stop. May be we did not wait long enough!

For example, in pseudocode, the program

While true:

continue

does not halt; rather, it goes on forever in an infinite loop.

On the other hand, the program

print "Hello World!"

halts very quickly.

The halting problem is famous because it was one of the first problems proven algorithmically undecidable. This means there is no algorithm which can be applied to any arbitrary program and input to decide whether the program stops when run with that input.

The Halting Problem is one of the simplest problems know to be unsolvable. You will note that the interesting problems involve loops.

Consider the following JavaScript program segments (algorithm):



The first program clearly is a one that will terminate after printing in an alert 10 lines of output. The second program alerts as many times as indicated by the input. The last program runs forever.

**Sketch of a proof that the Halting Problem is undecidable:**

This proof was devised by Alan Turing in 1936.

Suppose we have a solution to the halting problem called H. H takes two inputs:

1. a program P and

2. an input I for the program P.

H generates an output "***halt***" if H determines that P stops on input I or it outputs "***loop***" otherwise.



So now H can be revised to take P as both inputs (the program and its input) and H should be able to determine if P will halt on P as its input.

Let us construct a new, simple algorithm K that takes H's output as its input and does the following

1. If H outputs "***loop***" then K halts,

2. Otherwise H's output of "***halt***" causes K to loop forever.



That is, K will do the **opposite** of H's output.

**function K() {**

**if (H()=="loop"){**

**return;**

**} else {**

**while(true); //loop forever**

**}**

**}**

Since K is a program, let us use K as the input to K.



If H says that K halts then K itself would loop (that's how we constructed it). If H says that K loops then K will halt.

In either case H gives the wrong answer for K. **Thus H cannot work in all cases.**

We've shown that it is possible to construct an input that causes any solution H to fail.

Hence Proved!

**Post’s Correspondence Problem: (PCP)**

The Post’s Correspondence Problem is an undecidable decision problem that was Introduced by Emil Post in 1946.

Definition:

The input of the problem consists of two finite lists U= {u1, u2, ….., un} and V= {v1, v2,……., vn} of words over some alphabet Σ having at least two symbols.

A solution to this problem is a sequence of indices ik; 1<=k<= n, for all k, such that ui1 ui2 …………..uik = vi1 vi2 ………….. vik

We say i1, i2,…….ik is a solution to this instance of PCP.

Here, the decision problem is to decide whether such a solution exits or not.

Examples:   
Consider the following two lists:



A solution to this problem would be the sequence (3, 2, 3, 1), because

u3u2u3u1 = bba + ab + bba + a = bbaabbbaa

v3v2v3v1 = bb + aa + bb + baa = bbaabbbaa

Furthermore, since (3, 2, 3, 1) is a solution, so are all of its "repetitions", such as (3, 2, 3, 1, 3, 2, 3, 1), etc.; that is, when a solution exists, there are infinitely many solutions of this repetitive kind.

However, if the two lists had consisted of only u2,u3 and v2,v3, then there would have been no solution (because then no matching pair would have the same last letter, as must occur at the end of a solution).