Computational Complexity:

Complexity Theory is a central field of the theoretical foundations of Computer Science. It is concerned with the study of the *intrinsic complexity of computational tasks*. That is, a typical Complexity theoretic study looks at a task (or a class of tasks) and at the computational resources required to solve this task, rather than at a specific algorithm or algorithmic scheme.

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

Complexity Measure is a means of measuring the resource used during a computation. In case of Turing Machines, during any computation, various resources will be used, such as space and time. When estimating these resources, we are always interested in growth rates rather than absolute values.

A problem is regarded as inherently difficult if solving the problem requires a large amount of resources, whatever the algorithm used for solving it. The theory formalizes this intuition, by introducing mathematical models of computation to study these problems and quantifying the amount of resources needed to solve them, such as time and storage. Other complexity measures are also used, such as the amount of communication (used in communication complexity), the number of gates in a circuit (used in circuit complexity) and the number of processors (used in parallel computing). In particular, computational complexity theory determines the practical limits on what computers can and cannot do.

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:

Intractability is a technique for solving problems not to be solvable in polynomial time. The problems that cab be solved by any computational model, probably TM, using no more time then 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:

In computational complexity theory, a **complexity class** is a set of problems of related resource-based complexity. A typical complexity class has a definition of the form:

“The set of problems that can be solved by an abstract machine M using O(f(*n*)) of resource R, where *n* is the size of the input.”

For example, the **class NP** is the set of decision problems that can be solved by a non-deterministic Turing machine in polynomial time, while the **class P** is the set of decision problems that can be solved by a deterministic Turing machine in polynomial space.

The simpler complexity classes are defined by the following factors:

- *The type of computational problem:* The most commonly used problems are decision problems. However, complexity classes can be defined based on function problems, optimization problems, etc.

- *The model of computation:* The most common model of computation is the deterministic Turing machine, but many complexity classes are based on nondeterministic Turing machines, Boolean circuits etc.

- *The resource (or resources) that are being bounded and the bounds:* These two properties are usually stated together, such as "polynomial time", "logarithmic space", "constant depth", etc.

The set of problems that can be solved using polynomial time algorithm is regarded as **class P**. The problems that are verifiable in polynomial time constitute the **class NP**. The class of **NP complete** problems consists of those problems that are NP as well as they are *as hard as* any problem in NP. The main concern of studying NP completeness is to understand how hard the problem is. So if we can find some problem as NP complete then we try to solve the problem using methods like approximation, rather than searching for the faster algorithm for solving the problem exactly.

Class P: The class P is the set pf 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.

Note: *Since every deterministic TM is a non-deterministic TM having no choice of more than one moves, so P is subset of NP. However NP contains many problems that are not in P. The class* ***P*** *consists of all those decision problems that can be solved on a Deterministic Turing Machine in an amount of time that is polynomial in the size of the input; the class* ***NP*** *consists of all those decision problems whose positive solutions can be verified in polynomial time given the right information, or equivalently, whose solution can be found in polynomial time on a Non-deterministic Turing Machine.*

NP-Complete: In computational complexity theory, the complexity class **NP-complete** (abbreviated **NP-C** or **NPC**), is a class of problems having two properties:

→ It is in the set of NP (nondeterministic polynomial time) problems: Any given solution to the problem can be *verified* quickly (in polynomial time).

→It is also in the set of NP-hard problems: Any NP problem can be converted into this one by a transformation of the inputs in polynomial time.

Formally;

Let L be a language in NP, we say L is NP-Complete if the following statements are true about L;

→L is in class NP

→ For every language L1 in NP, there is a polynomial time reduction of L1 to L.

Once we have some NP-Complete problem, we can prove a new problem to be NP-Complete by reducing some known NP-Complete problem to it using polynomial time reduction.

It is not known whether every problem in NP can be quickly solved—this is called the P = NP problem. But if *any single problem* in NP-complete can be solved quickly, then *every problem in NP* can also be quickly solved, because the definition of an NP-complete problem states that every problem in NP must be quickly reducible to every problem in NP-complete (that is, it can be reduced in polynomial time).

Because of this, it is often said that the NP-complete problems are *harder* or *more difficult* than NP problems in general.

Some of the properties of NP-complete problems are:

- No polynomial time algorithms has been found for any of them.
- It is no established that polynomial time algorithm for these problems do not exist.
- If polynomial time algorithm is found for any one of them, there will be polynomial time algorithm for all of them.
- If it can be proved that no polynomial time algorithm exists for any of them, then it will not exist for every one of them.

Problems:

Abstract Problems:

Abstract problem A is binary relation on set I of problem instances, and the set S of problem solutions. For e.g. Minimum spanning tree of a graph G can be viewed as a pair of the given graph G and MST graph T.

Decision Problems:

Decision problem D is a problem that has an answer as either "true", "yes", "1" or "false", "no", "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.

Function Problems:

In computational complexity theory, a **function problem** is a computational problem where a single output (of a total function) is expected for every input, but the output is more complex than that of a decision problem, that is, it isn't just YES or NO. Notable examples include the *Traveling salesman problem*, which asks for the route taken by the salesman, and the *Integer factorization problem*, which asks for the list of factors. Function problems can be sorted into complexity classes in the same way as decision problems. For example FP is the set of function problems which can be solved by a deterministic

Turing machine in polynomial time, and FNP is the set of function problems which can be solved by a non-deterministic Turing machine in polynomial time. For all function problems in which the solution is polynomially bounded, there is an analogous decision problem such that the function problem can be solved by polynomial-time Turing reduction to that decision problem.

Encoding:

Encoding of a set S is a function e from S to the set of binary strings. With the help of encoding, we define **concrete problem** as a problem with problem instances as the set of binary strings i.e. if we encode the abstract problem, then the resulting encoded problem is concrete problem. So, encoding as a concrete problem assures that every encoded problem can be regarded as a language i.e. subset of {0,1}*.

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 fist 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*. There are many different type of reductions, based on the method of reduction, such as Cook reductions, Karp reductions and Levin reductions, and the bound on the complexity of reductions, such as *polynomial-time reductions* or *log-space reductions*.

The most commonly used reduction is a polynomial-time reduction. This means that the reduction process takes polynomial time. *For example, the problem of squaring an integer can be reduced to the problem of multiplying two integers. This means an algorithm for multiplying two integers can be used to square an integer*. Indeed, this can be done by giving the same input to both inputs of the multiplication algorithm. Thus we see that squaring is not more difficult than multiplication, since squaring can be reduced to multiplication.

This motivates the concept of a problem being hard for a complexity class. A problem *X* is *hard* for a class of problems *C* if every problem in *C* can be reduced to *X*. Thus no problem in *C* is harder than *X*, since an algorithm for *X* allows us to solve any problem in *C*. Of course, the notion of hard problems depends on the type of reduction being used. For complexity classes larger than P, polynomial-time reductions are commonly used. In particular, the set of problems that are hard for NP is the set of NP-hard problems.

Circuit Satisfaibility:

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 A_x be the circuit obtained from A by setting the x-input wire values according to the specific string x. The construction of A_x from x is our reduction function. If x is a yes instance of V, then the certificate y for x gives satisfying assignments for A_x. Conversely, if A_x 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.

Note:

(Refer Chapter "NP-Completeness" of the Book "Introduction to Algorithms": By Cormen, Leiserson, Rivest and Stein. It's a reference book of DA!!!!)

(From Page 966 to…., In 2nd Ed.) (You will find detail concepts of SAT, BSAT)

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.

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 lest two symbols. A solution to this problem is a sequence of indices ik; 1<=k<= n, for all k, such that

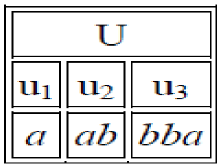
$u_{i1}\ u_{i2}\ \ldots\ldots\ldots\ldots.\ u_{ik} = v_{i1}\ v_{i2}\ \ldots\ldots\ldots\ldots.\ v_{ik}$

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

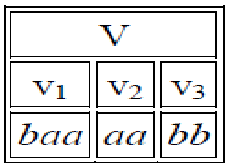
Here, the decision problem is to decide whether such a solution exits or not.

Examples:

Consider the following two lists:



U		
u_1	u_2	u_3
a	*ab*	*bba*



V		
v_1	v_2	v_3
baa	*aa*	*bb*

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

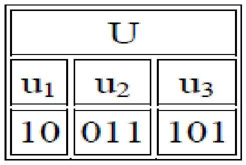
u3u2u3u1 = *bba* + *ab* + *bba* + *a* = *bbaabbbaa*

v3v2v3v1 = *bb* + *aa* + *bb* + *baa* = *bbaabbbaa*

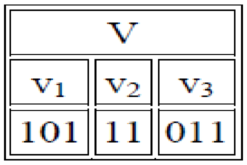
Furthermore, since (3, 2, 3, 1) is a solution, so are all of its "repetitions", such as (3, 2, 3, 1, 3, 2, 3, 1), etc.; that is, when a solution exists, there are infinitely many solutions of this repetitive kind.

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).

Consider another example with two lists as below



U		
u_1	u_2	u_3
10	011	101



V		
v_1	v_2	v_3
101	11	011

For this instance of PCP, there is no solution !!!

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 consists 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 can not 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

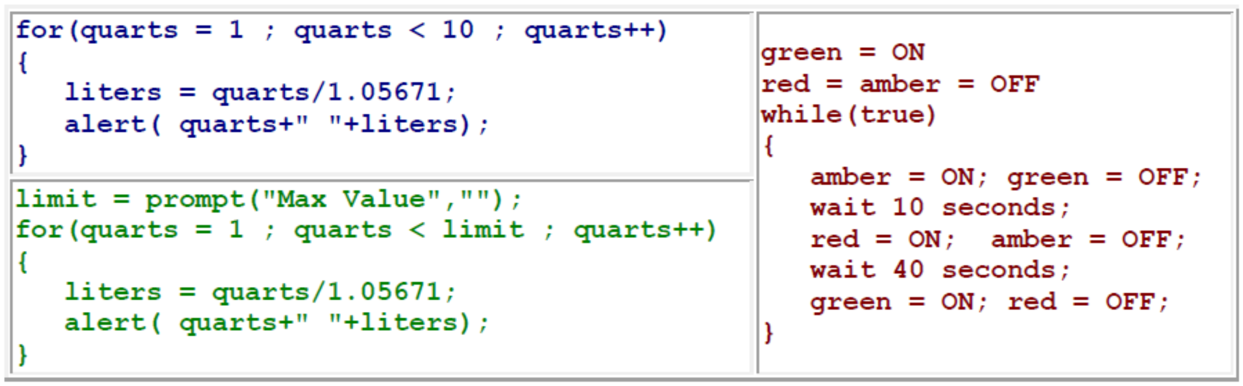
print "Hello World!"

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):



```
for(quarts = 1 ; quarts < 10 ; quarts++)
{
   liters = quarts/1.05671;
   alert( quarts+" "+liters);
}
```

```
limit = prompt("Max Value","");
for(quarts = 1 ; quarts < limit ; quarts++)
{
   liters = quarts/1.05671;
   alert( quarts+" "+liters);
}
```

```
green = ON
red = amber = OFF
while(true)
{
   amber = ON; green = OFF;
   wait 10 seconds;
   red = ON;  amber = OFF;
   wait 40 seconds;
   green = ON; red = OFF;
}
```

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:

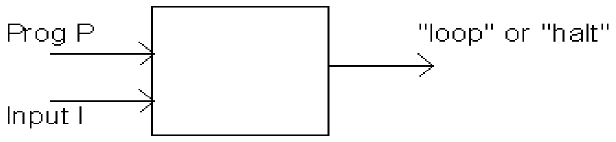
This proof was devised by Alan Turing in 1936.

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.



Note: When an algorithm is coded, it is expressed as a string of characters which can also be interpreted as a sequence of numbers (binary). We can treat the program as data and therefore a program can be thought of as input.

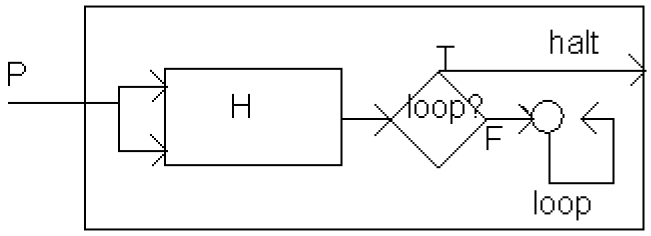
For example, compilers take programs as input and generate machine code as output. Netscape takes a Javascript program and generates output.

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.

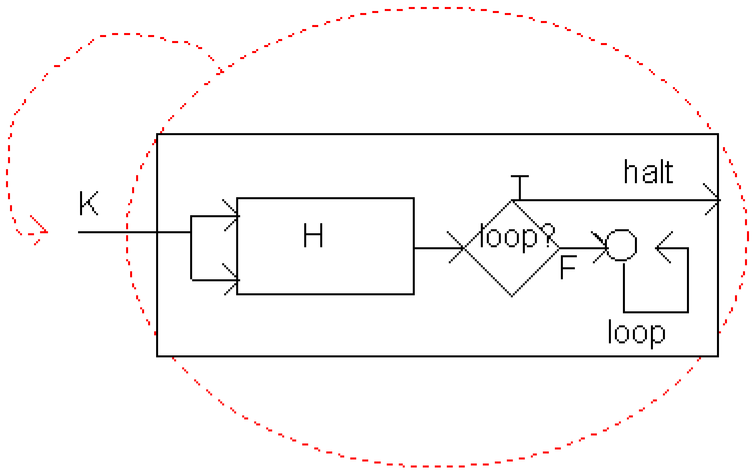


That is, K will do the **opposite** of H's output.

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!