→A nondeterministic algorithm is said to be ***nondeterministic polynomial*** if the time complexity of its verification stage is polynomial.

→**Tractable Problems:** Problems that can be solved in polynomial time are called ***tractable***.
Intractable Problems: Problems that cannot be solved in polynomial time are called ***intractable***.

→Some decision problems cannot be solved at all by any algorithm. Such problems are called ***undecidable***, as opposed to ***decidable*** problems that can be solved by an algorithm.
→A famous *example of an undecidable* problem was given by Alan Turing in 1936. It is called the ***halting problem***: given a computer program and an input to it, determine whether the program will halt on that input or continue working indefinitely on it.

REDUCIBILITY:

→ A decision problem D_1 is said to be ***polynomially reducible*** to a decision problem D_2 *(also written as* $D_1 \propto D_2$*)*, if there exists a function *t* that transforms instances of D_1 into instances of D_2 such that:

1. *t* maps all *Yes* instances of D_1 to *Yes* instances of D_2 and all *No* instances of D_1 to *No* instances of D_2.

2. *t* is computable by a polynomial time algorithm.

→The definition for $D_1 \propto D_2$ immediately implies that if D_2 can be solved in *polynomial time*, then D_1 can also be solved in *polynomial time*. In other words, if D_2 has a deterministic polynomial time algorithm, then D_1 can also have a deterministic polynomial time algorithm.

Based on this, we can also say that, if D_2 is easy, then D_1 can also be easy. In other words, D_1 is as easy as D_2. Easiness of D_2 proves the easiness of D_1.

→But, here we mostly focus on showing *how hard a problem is* rather than how easy it is, by using the contra positive meaning of the reduction as follows:

$D_1 \propto D_2$ implies that if D_1 cannot be solved in *polynomial time*, then D_2 also cannot be solved in *polynomial time*. In other words, if D_1 does not have a deterministic polynomial time algorithm, then D_2 also can not have a deterministic polynomial time algorithm.

We can also say that, if D_1 is hard, then D_2 can also be hard. In other words, D_2 is as hard as D_1.

→To show that problem D_1 (i.e., new problem) is at least as hard as problem D_2 (i.e., known problem), we need to reduce D_2 to D_1 (not D_1 to D_2).

→Reducibility ($\propto$) is a transitive relation, that is, if $D_1 \propto D_2$ and $D_2 \propto D_3$ then $D_1 \propto D_3$.

NP-HARD CLASS:

→A problem 'L' is said to be NP-Hard iff every problem in NP reduces to 'L'

(or)

→ A problem 'L' is said to be NP-Hard if it is as hard as any problem in NP.

(or)

→A problem 'L' is said to be NP-Hard iff SAT reduces to 'L'.

Since SAT is a known NP-Hard problem, every problem in NP can be reduced to SAT. So, if SAT reduces to L, then every problem in NP can be reduced to 'L'.

Ex: SAT and Clique problems.

→An NP-Hard problem *need not be* NP problem.

Ex: *Halting Problem* is NP-Hard **but not** NP.

NP-COMPLETE CLASS:

→A problem 'L' is said to be NP-Complete if 'L' is NP-Hard and L ∈ *NP*.

→These are the hardest problems in NP set.

Ex: SAT and Clique problems.

Showing that a decision problem is *NP*-complete:

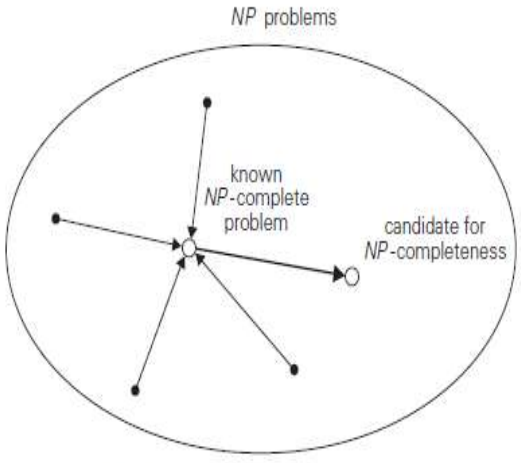
It can be done in two steps:

Step1:

Show that the problem in question is in *NP*; i.e., a randomly generated string can be checked in polynomial time to determine whether or not it represents a solution to the problem. Typically, this step is easy.

Step2:

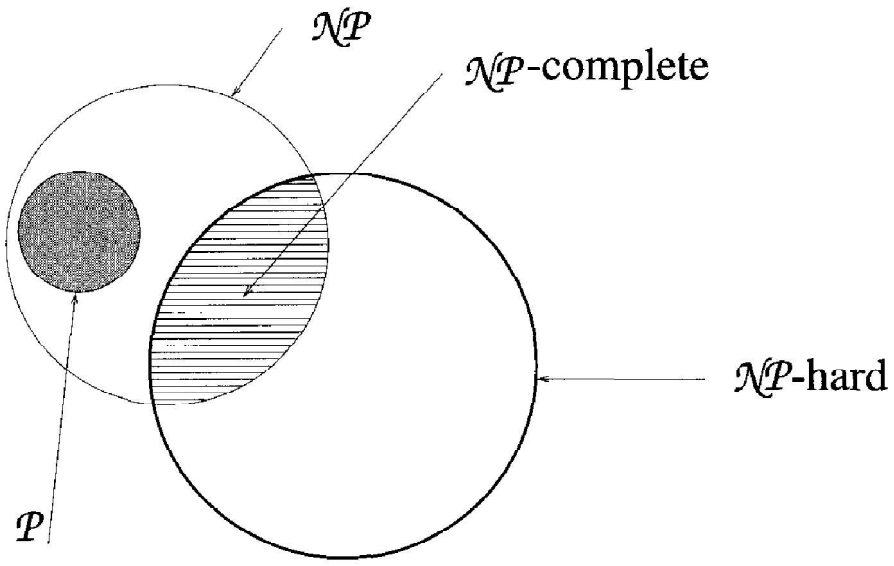
Show that the problem in question is NP-Hard also. That means, show that every problem in *NP* is reducible to the problem in question, in polynomial time. Because of the transitivity of polynomial reduction, this step can be done by showing that a known *NP*-complete problem can be transformed into the problem in question, in polynomial time, as depicted in the figure below.



Proving *NP*-completeness by reduction.

→The definition of *NP*-completeness immediately implies that if there exists a polynomial-time algorithm for just one *NP*-Complete problem, then every problem in *NP* can also have a polynomial time algorithm, and hence *P* = *NP*.

Relationship among P, NP, NP-Hard and NP-Complete Classes:



COOK'S THEOREM:

→Cook's theorem can be stated as follows.

(1) SAT is NP-Complete.

(or)

(2) If *SAT* is in *P* then *P* = *NP*. That means, if there is a polynomial time algorithm for *SAT*, then there is a polynomial time algorithm for every other problem in *NP*.

(or)

(3) *SAT* is in *P* iff *P* = *NP*.

Application of Cook's Theorem:

A new problem 'L' can be proved NP-Complete by reducing SAT to 'L' in polynomial time, provided 'L' is NP problem. Since SAT is

NP-Complete, every problem in NP can be reduced to SAT. So, once SAT reduces to ‘L’, then every problem in NP can be reduced to ‘L’ proving that ‘L’ is NP-Hard. Since ‘L’ is NP also, we can say that ‘L’ is NP-Complete.

Example Problem: Prove that Clique problem is NP-Complete.

(OR)

Reduce SAT problem to Clique problem.

Solution: See the video at https://www.youtube.com/watch?v=qZs767KQcvE
