CHAPTER

20

COMPLEXITY CLASSES

20.1 Introduction

In the previous chapters we have solved problems of different complexities. Some algorithms have lower rates of growth while others have higher rates of growth. The problems with lower rates of growth are called *easy* problems (or *easy solved problems*) and the problems with higher rates of growth are called *hard* problems (or *hard solved problems*). This classification is done based on the running time (or memory) that an algorithm takes for solving the problem.

Time Complexity	Name	Example	Problems
O(1)	Constant	Adding an element to the front of a linked list	Easy solved problems
O($logn$)	Logarithmic	Finding an element in a binary search tree	
O(n)	Linear	Finding an element in an unsorted array	
O($nlogn$)	Linear Logarithmic	Merge sort	
O(n^2)	Quadratic	Shortest path between two nodes in a graph	
O(n^3)	Cubic	Matrix Multiplication	
O(2^n)	Exponential	The Towers of Hanoi problem	Hard solved problems
O($n!$)	Factorial	Permutations of a string	

There are lots of problems for which we do not know the solutions. All the problems we have seen so far are the ones which can be solved by computer in deterministic time. Before starting our discussion let us look at the basic terminology we use in this chapter.

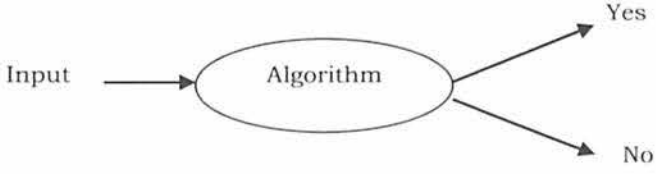
20.2 Polynomial/Exponential Time

Exponential time means, in essence, trying every possibility (for example, backtracking algorithms) and they are very slow in nature. Polynomial time means having some clever algorithm to solve a problem, and we don't try every possibility. Mathematically, we can represent these as:

- Polynomial time is O(n^k), for some k.
- Exponential time is O(k^n), for some k.

20.3 What is a Decision Problem?

A decision problem is a question with a *yes/no* answer and the answer depends on the values of input. For example, the problem "Given an array of n numbers, check whether there are any duplicates or not?" is a decision problem. The answer for this problem can be either *yes* or *no* depending on the values of the input array.



20.4 Decision Procedure

For a given decision problem let us assume we have given some algorithm for solving it. The process of solving a given decision problem in the form of an algorithm is called a *decision procedure* for that problem.

20.5 What is a Complexity Class?

In computer science, in order to understand the problems for which solutions are not there, the problems are divided into classes and we call them as complexity classes. In complexity theory, a *complexity class* is a set of problems with related complexity. It is the branch of theory of computation that studies the resources required during computation to solve a given problem.

The most common resources are time (how much time the algorithm takes to solve a problem) and space (how much memory it takes).

20.6 Types of Complexity Classes

P Class

The complexity class P is the set of decision problems that can be solved by a deterministic machine in polynomial time (P stands for polynomial time). P problems are a set of problems whose solutions are easy to find.

NP Class

The complexity class NP (NP stands for non-deterministic polynomial time) is the set of decision problems that can be solved by a non-deterministic machine in polynomial time. NP class problems refer to a set of problems whose solutions are hard to find, but easy to verify.

For better understanding let us consider a college which has 500 students on its roll. Also, assume that there are 100 rooms available for students. A selection of 100 students must be paired together in rooms, but the dean of students has a list of pairings of certain students who cannot room together for some reason.

The total possible number of pairings is too large. But the solutions (the list of pairings) provided to the dean, are easy to check for errors. If one of the prohibited pairs is on the list, that's an error. In this problem, we can see that checking every possibility is very difficult, but the result is easy to validate.

That means, if someone gives us a solution to the problem, we can tell them whether it is right or not in polynomial time. Based on the above discussion, for NP class problems if the answer is *yes*, then there is a proof of this fact, which can be verified in polynomial time.

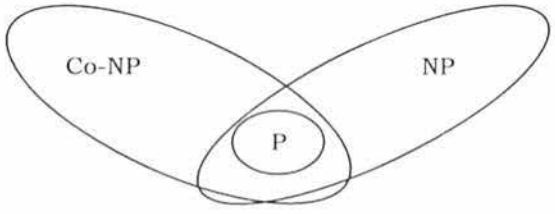
Co-NP Class

$Co-NP$ is the opposite of NP (complement of NP). If the answer to a problem in $Co-NP$ is *no*, then there is a proof of this fact that can be checked in polynomial time.

P	Solvable in polynomial time
NP	*Yes* answers can be checked in polynomial time
$Co-NP$	*No* answers can be checked in polynomial time

Relationship between P, NP and Co-NP

Every decision problem in P is also in NP. If a problem is in P, we can verify YES answers in polynomial time. Similarly, any problem in P is also in $Co-NP$.



One of the important open questions in theoretical computer science is whether or not $P = NP$. Nobody knows. Intuitively, it should be obvious that $P \neq NP$, but nobody knows how to prove it.

Another open question is whether NP and $Co - NP$ are different. Even if we can verify every YES answer quickly, there's no reason to think that we can also verify NO answers quickly.

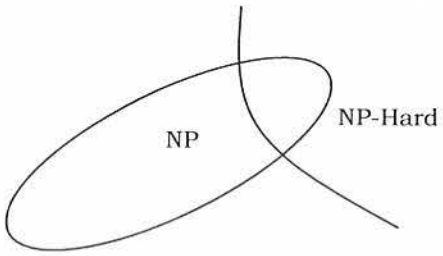
It is generally believed that $NP \neq Co - NP$, but again nobody knows how to prove it.

NP-hard Class

It is a class of problems such that every problem in NP reduces to it. All NP-hard problems are not in NP, so it takes a long time to even check them. That means, if someone gives us a solution for NP-hard problem, it takes a long time for us to check whether it is right or not.

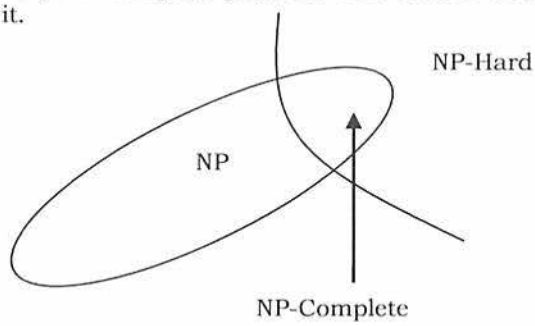
A problem K is NP-hard indicates that if a polynomial-time algorithm (solution) exists for K then a polynomial-time algorithm for every problem is NP. Thus:

K is NP-hard implies that if K can be solved in polynomial time, then $P = NP$



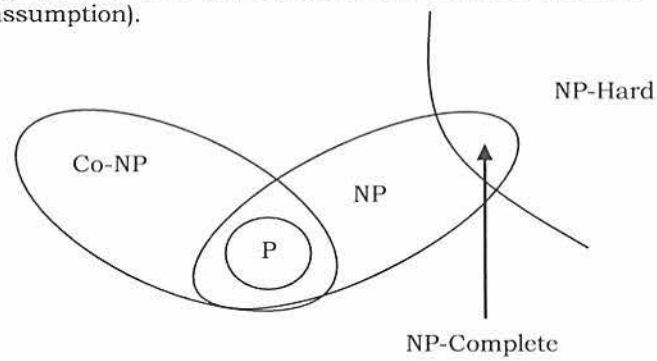
NP-complete Class

Finally, a problem is NP-complete if it is part of both NP-hard and NP. NP-complete problems are the hardest problems in NP. If anyone finds a polynomial-time algorithm for one NP-complete problem, then we can find polynomial-time algorithm for every NP-complete problem. This means that we can check an answer fast and every problem in NP reduces to it.



Relationship between P, NP Co-NP, NP-Hard and NP-Complete

From the above discussion, we can write the relationships between different components as shown below (remember, this is just an assumption).



The set of problems that are NP-hard is a strict superset of the problems that are NP-complete. Some problems (like the halting problem) are NP-hard, but not in NP. NP-hard problems might be impossible to solve in general. We can tell the difference in difficulty between NP-hard and NP-complete problems because the class

NP includes everything easier than its "toughest" problems – if a problem is not in *NP*, it is harder than all the problems in *NP*.

Does P==NP?

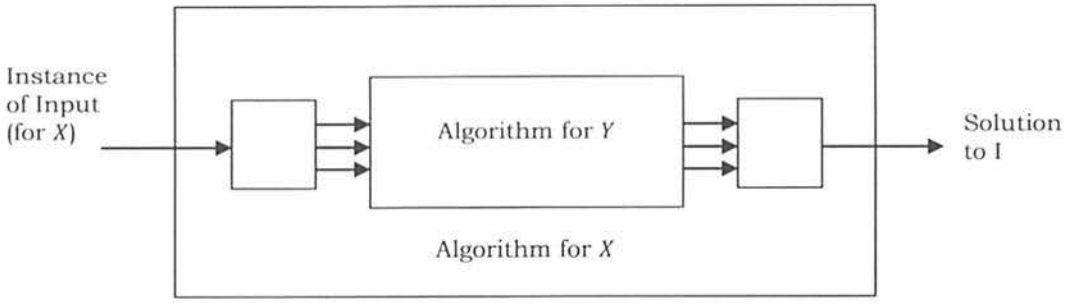
If $P = NP$, it means that every problem that can be checked quickly can be solved quickly (remember the difference between checking if an answer is right and actually solving a problem).

This is a big question (and nobody knows the answer), because right now there are lots of *NP*-complete problems that can't be solved quickly. If $P = NP$, that means there is a way to solve them fast. Remember that "quickly" means not trial-and-error. It could take a billion years, but as long as we didn't use trial and error, it was quick. In future, a computer will be able to change that billion years into a few minutes.

20.7 Reductions

Before discussing reductions, let us consider the following scenario. Assume that we want to solve problem X but feel it's very complicated. In this case what do we do?

The first thing that comes to mind is, if we have a similar problem to that of X (let us say Y), then we try to map X to Y and use $Y's$ solution to solve X also. This process is called reduction.



In order to map problem X to problem Y, we need some algorithm and that may take linear time or more. Based on this discussion the cost of solving problem X can be given as:

$$Cost\ of\ solving\ X = Cost\ of\ solving\ Y + Reduction\ time$$

Now, let us consider the other scenario. For solving problem X, sometimes we may need to use $Y's$ algorithm (solution) multiple times. In that case,

$$Cost\ of\ solving\ X = Number\ of\ Times * Cost\ of\ solving\ X + Reduction\ time$$

The main thing in *NP*-Complete is reducibility. That means, we reduce (or transform) given *NP*-Complete problems to other known *NP*-Complete problem. Since the *NP*-Complete problems are hard to solve and in order to prove that given *NP*-Complete problem is hard, we take one existing hard problem (which we can prove is hard) and try to map given problem to that and finally we prove that the given problem is hard.

Note: It's not compulsory to reduce the given problem to known hard problem to prove its hardness. Sometimes, we reduce the known hard problem to given problem.

Important NP-Complete Problems (Reductions)

Satisfiability Problem: A boolean formula is in *conjunctive normal form* (CNF) if it is a conjunction (AND) of several clauses, each of which is the disjunction (OR) of several literals, each of which is either a variable or its negation. For example: $(a \vee b \vee c \vee d \vee e) \wedge (b \vee \sim c \vee \sim d) \wedge (\sim a \vee c \vee d) \wedge (a \vee \sim b)$

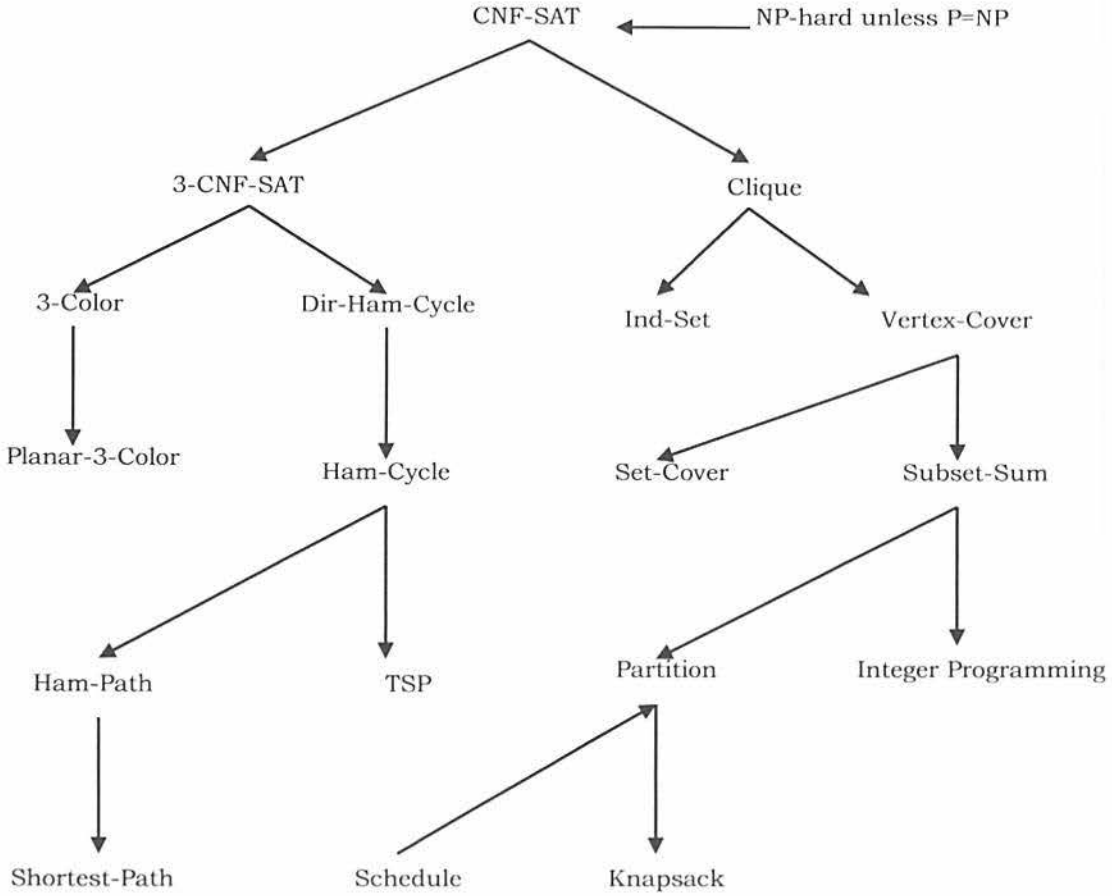
A 3-CNF formula is a CNF formula with exactly three literals per clause. The previous example is not a 3-CNF formula, since its first clause has five literals and its last clause has only two.

2-SAT Problem: 3-SAT is just SAT restricted to 3-CNF formulas: Given a 3-CNF formula, is there an assignment to the variables so that the formula evaluates to TRUE?

2-SAT Problem: 2-SAT is just SAT restricted to 2-CNF formulas: Given a 2-CNF formula, is there an assignment to the variables so that the formula evaluates to TRUE?

Circuit-Satisfiability Problem: Given a boolean combinational circuit composed of AND, OR and NOT gates, is it satisfiable?. That means, given a boolean circuit consisting of AND, OR and NOT gates properly connected by

wires, the Circuit-SAT problem is to decide whether there exists an input assignment for which the output is TRUE.



Hamiltonian Path Problem (Ham-Path): Given an undirected graph, is there a path that visits every vertex exactly once?

Hamiltonian Cycle Problem (Ham-Cycle): Given an undirected graph, is there a cycle (where start and end vertices are same) that visits every vertex exactly once?

Directed Hamiltonian Cycle Problem (Dir-Ham-Cycle): Given a directed graph, is there a cycle (where start and end vertices are same) that visits every vertex exactly once?

Travelling Salesman Problem (TSP): Given a list of cities and their pair-wise distances, the problem is to find the shortest possible tour that visits each city exactly once.

Shortest Path Problem (Shortest-Path): Given a directed graph and two vertices s and t, check whether there is a shortest simple path from s to t.

Graph Coloring: A k-coloring of a graph is to map one of k 'colors' to each vertex, so that every edge has two different colors at its endpoints. The graph coloring problem is to find the smallest possible number of colors in a legal coloring.

3-Color problem: Given a graph, is it possible to color the graph with 3 colors in such a way that every edge has two different colors?

Clique (also called complete graph): Given a graph, the *CLIQUE* problem is to compute the number of nodes in its largest complete subgraph. That means, we need to find the maximum subgraph which is also a complete graph.

Independent Set Problem (Ind_Set): Let G be an arbitrary graph. An independent set in G is a subset of the vertices of G with no edges between them. The maximum independent set problem is the size of the largest independent set in a given graph.

Vertex Cover Problem (Vertex-Cover): A vertex cover of a graph is a set of vertices that touches every edge in the graph. The vertex cover problem is to find the smallest vertex cover in a given graph.

Subset Sum Problem (Subset-Sum): Given a set S of integers and an integer T, determine whether S has a subset whose elements sum to T.

Integer Programming: Given integers b_i, a_{ij} find 0/1 variables x_i that satisfy a linear system of equations.

$$\sum_{j=1}^{N} a_{ij}x_j = b_i \quad 1 \le i \le M$$
$$x_j \in \{0,1\}\ 1 \le j \le N$$

In the figure, arrows indicate the reductions. For example, Ham-Cycle (Hamiltonian Cycle Problem) can be reduced to CNF-SAT. Same is the case with any pair of problems. For our discussion, we can ignore the reduction process for each of the problems. There is a theorem called *Cook's Theorem* which proves that Circuit satisfiability problem is NP-hard. That means, Circuit satisfiability is a known *NP*-hard problem.

Note: Since the problems below are *NP*-Complete, they are *NP* and *NP*-hard too. For simplicity we can ignore the proofs for these reductions.

20.8 Complexity Classes: Problems & Solutions

Problem-1 What is a quick algorithm?

Solution: A quick algorithm (solution) means not trial-and-error solution. It could take a billion years, but as long as we do not use trial and error, it is efficient. Future computers will change those billion years to a few minutes.

Problem-2 What is an efficient algorithm?

Solution: An algorithm is said to be efficient if it satisfies the following properties:

- Scale with input size.
- Don't care about constants.
- Asymptotic running time: polynomial time.

Problem-3 Can we solve all problems in polynomial time?

Solution: **No**. The answer is trivial because we have seen lots of problems which take more than polynomial time.

Problem-4 Are there any problems which are *NP*-hard?

Solution: By definition, *NP*-hard implies that it is very hard. That means it is very hard to prove and to verify that it is hard. Cook's Theorem proves that Circuit satisfiability problem is *NP*-hard.

Problem-5 For 2-SAT problem, which of the following are applicable?
(a) *P* (b) *NP* (c) *CoNP* (d) *NP*-Hard
(e) *CoNP*-Hard (f) *NP*-Complete (g) *CoNP*-Complete

Solution: 2-SAT is solvable in poly-time. So it is *P*, *NP*, and *CoNP*.

Problem-6 For 3-SAT problem, which of the following are applicable?
(a) *P* (b) *NP* (c) *CoNP* (d) *NP*-Hard
(e) *CoNP*-Hard (f) *NP*-Complete (g) *CoNP*-Complete

Solution: 3-SAT is NP-complete. So it is NP, NP-Hard, and NP-complete.

Problem-7 For 2-Clique problem, which of the following are applicable?
(a) *P* (b) *NP* (c) *CoNP* (d) *NP*-Hard
(e) *CoNP*-Hard (f) *NP*-Complete (g) *CoNP*-Complete

Solution: 2-Clique is solvable in poly-time (check for an edge between all vertex-pairs in $O(n^2)$ time). So it is *P*, *NP*, and *CoNP*.

Problem-8 For 3-Clique problem, which of the following are applicable?
(a) *P* (b) *NP* (c) *CoNP* (d) *NP*-Hard
(e) *CoNP*-Hard (f) *NP*-Complete (g) *CoNP*-Complete

Solution: 3-Clique is solvable in poly-time (check for a triangle between all vertex-triplets in $O(n^3)$ time). So it is *P*, *NP*, and *CoNP*.

Problem-9 Consider the problem of determining. For a given boolean formula, check whether every assignment to the variables satisfies it. Which of the following is applicable?
(a) *P* (b) *NP* (c) *CoNP* (d) *NP*-Hard
(e) CoNP-Hard (f) *NP*-Complete (g) *CoNP*-Complete

Solution: Tautology is the complimentary problem to Satisfiability, which is NP-complete, so Tautology is *CoNP*-complete. So it is *CoNP*, *CoNP*-hard, and *CoNP*-complete.

Problem-10 Let S be an NP-complete problem and Q and R be two other problems not known to be in NP. Q is polynomial time reducible to S and S is polynomial-time reducible to R. Which one of the following statements is true?
(a) R is NP-complete (b) R is NP-hard (c) Q is NP-complete (d) Q is NP-hard.

Solution: R is NP-hard (b).

Problem-11 Let A be the problem of finding a Hamiltonian cycle in a graph $G = (V, E)$, with $|V|$ divisible by 3 and B the problem of determining if Hamiltonian cycle exists in such graphs. Which one of the following is true?
(a) Both A and B are NP-hard (b) A is NP-hard, but B is not
(c) A is NP-hard, but B is not (d) Neither A nor B is NP-hard

Solution: Both A and B are NP-hard (a).

Problem-12 Let A be a problem that belongs to the class NP. State which of the following is true?
(a) There is no polynomial time algorithm for A.
(b) If A can be solved deterministically in polynomial time, then $P = NP$.
(c) If A is NP-hard, then it is NP-complete.
(d) A may be undecidable.

Solution: If A is NP-hard, then it is NP-complete (c).

Problem-13 Suppose we assume $Vertex - Cover$ is known to be NP-complete. Based on our reduction, can we say $Independent - Set$ is NP-complete?

Solution: Yes. This follows from the two conditions necessary to be NP-complete:

- Independent Set is in NP, as stated in the problem.
- A reduction from a known NP-complete problem.

Problem-14 Suppose *Independent Set* is known to be NP-complete. Based on our reduction, is *Vertex Cover* NP-complete?

Solution: No. By reduction from Vertex-Cover to Independent-Set, we do not know the difficulty of solving Independent-Set. This is because Independent-Set could still be a much harder problem than Vertex-Cover. We have not proved that.

Problem-15 The class of NP is the class of languages that cannot be accepted in polynomial time. Is it true? Explain.

Solution:

- The class of NP is the class of languages that can be *verified* in *polynomial time*.
- The class of P is the class of languages that can be *decided* in *polynomial time*.
- The class of P is the class of languages that can be *accepted* in *polynomial time*.

$P \subseteq NP$ and "languages in P can be accepted in polynomial time", the description "languages in NP cannot be accepted in polynomial time" is wrong.

The term NP comes from nondeterministic polynomial time and is derived from an alternative characterization by using nondeterministic polynomial time Turing machines. It has nothing to do with "cannot be accepted in polynomial time".

Problem-16 Different encodings would cause different time complexity for the same algorithm. Is it true?

Solution: True. The time complexity of the same algorithm is different between unary encoding and binary encoding. But if the two encodings are polynomially related (e.g. base 2 & base 3 encodings), then changing between them will not cause the time complexity to change.

Problem-17 If P = NP, then NPC (NP Complete) $\subseteq$ P. Is it true?

Solution: True. If P = NP, then for any language L $\in$ NP C (1) L $\in$ NPC (2) L is NP-hard. By the first condition, L $\in$ NPC $\subseteq$ NP = P $\Rightarrow$ NPC $\subseteq$ P.

Problem-18 If NPC $\subseteq$ P, then P = NP. Is it true?

Solution: True. All the NP problem can be reduced to arbitrary NPC problem in polynomial time, and NPC problems can be solved in polynomial time because NPC $\subseteq$ P. $\Rightarrow$ NP problem solvable in polynomial time $\Rightarrow$ NP $\subseteq$ P and trivially P $\subseteq$ NP implies NP = P.