DESIGN & ANALYSIS OF ALGORITHMS

UNIT – VI

6.1. Introduction to NP-Hard and NP-Complete problems

6.2. Basic concepts of non deterministic algorithms

6.3. Definitions of NP-Hard Classes

6.4. Definitions of NP-Complete Classes

6.5. Modular Arithmetic

Introduction:

In Computer Science, many problems are solved where the objective is to maximize or minimize some values, whereas in other problems we try to find whether there is a solution or not.

Hence, the problems can be categorized as follows

1. Optimization Problems are those for which the objective is to maximize or minimize some values.

Examples:

- **Finding the minimum number of colors needed to color a given graph.**
- **Finding the shortest path between two vertices in a graph.**

❖ **This group consists of the problems that can be solved in polynomial time.**

❖ **Example: Searching of an elements from the list O(log n), sorting of elements O(log n).**

2. Decision Problems **There are many problems for which the answer is a Yes or a No. These types of problems are known as Decision Problems.**

Example:

- **Whether a given graph can be colored by only 4-colors.**
- **Finding Hamiltonian cycle in a graph is not a decision problem, whereas checking a graph is Hamiltonian or not is a decision problem.**

This group consists of problems that can be solved in Non-deterministic Polynomial time.

Example: Knapsack problem $O(2^{n/2})$ and Travelling Salesperson problem $O(n^2 2^n)$

What is Language?

•**Every decision problem can have only two answers, yes or no. Hence, a decision problem may belong to a language if it provides an answer 'yes' for a specific input. A language is the totality of inputs for which the answer is Yes.**

•**For input size n, if worst-case time complexity of an algorithm is $O(n^k)$, where k is a constant, the algorithm is a polynomial time algorithm.**

P Class Problems

➢**The class P consists of those problems that are solvable in Polynomial time, i.e. these problems can be solved in time $O(n^k)$ in worst-case, where k is constant. These problems are called Tractable.**

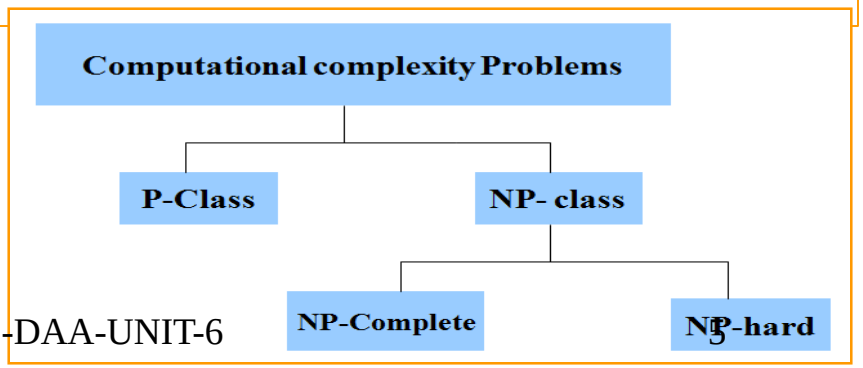
➢**Formally, an algorithm is polynomial time algorithm, if there exists a polynomial p(n) such that the algorithm can solve any instance of size n in a time O(p(n)).**

NP Class Problems

- **Now there are a lot of programs that don't (necessarily) run in polynomial time on a regular computer, but do run in polynomial time on a nondeterministic Turing machine. These programs solve problems in NP, which stands for Non Deterministic Polynomial time. or**

An equivalent way to define NP is by pointing to the problems that can be verified in polynomial time. This means there is not necessarily a polynomial-time way to find a solution, but once you have a solution it only takes polynomial time to verify that it is correct.

The NP class problems can be further categorized into NP-complete and NP hard problems.



NP Complete & NP Hard Class Problems

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.**

- **Another way:**

Take two NP problems A and B.

Reducibility: **If we can convert one instance of a problem A into problem B (NP problem) then it means that A is reducible to B.**

NP-hard: **Now suppose we found that A is reducible to B, then it means that B is at least as hard as A.**

NP-Complete: **The group of problems which are both in NP and NP-hard are known as NP-Complete problem.**

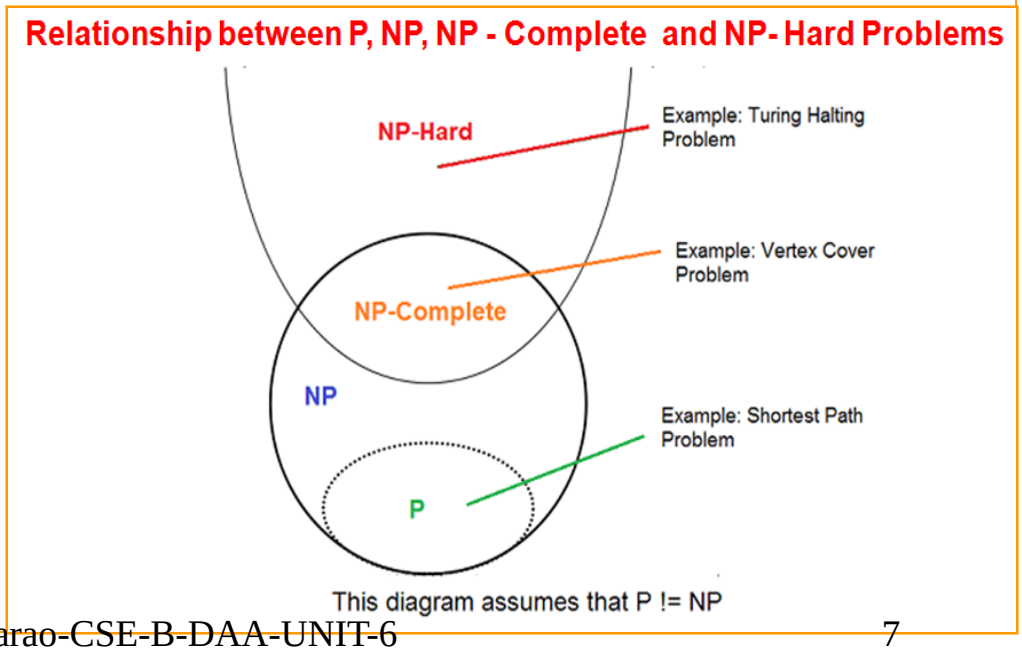
Now suppose we have a NP-Complete problem R and it is reducible to Q then Q is at least as hard as R and since R is an NP-hard problem. therefore Q will also be at least NP-hard , it may be NP-complete also.

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**



Non Deterministic Algorithm

•The algorithm in which every operation is uniquely defined is called Deterministic Algorithm.

•The algorithm in which every operation may not have unique result, rather there can be specified set of possibilities for every operation. Such an algorithm is called Non Deterministic Algorithm.

Non deterministic means that no particular rule is followed to make a guess.

•The non deterministic algorithm is a two stage algorithm

Non deterministic ("Guessing") stage: **generate an arbitrary string that can be thought of as a candidate solution.**

Deterministic ("Verification") stage: **In this stage it takes as input the candidate solution and the instance to the problem and returns 'yes' if the candidate solution represents actual solution.**

```
1. Algorithm Non_Determin( )
// A[1:n] is a set of elements. We have to determine the index i of A at which element x is located.
2. {   // The following for-loop is the guessing stage
3.        for i=1 to n do
4.           A[i] := choose(i);   // Next is the verification (deterministic) stage
5.        if (A[i] = x) then
6.        {
7.            write(i);
8.            success();
9.        }
10.       write(0);
11.       fail();
12. }
```

In the above given non deterministic algorithm there are three functions used-

1)Choose **– arbitrarily chooses one of the element from given input set.**

2)Fail **– indicates the unsuccessful completion.**

3)Success **– indicates successful completion**

The algorithm is of non deterministic complexity O(1), when A is not ordered then the deterministic search algorithm has a complexity Ω(n)

Cook's Theorem :

Theorem-1

If a set **S** of strings is accepted by some non-deterministic Turing machine within polynomial time, then **S** is P-reducible to {DNF tautologies}.

Theorem-2

The following sets are P-reducible to each other in pairs (and hence each has the same polynomial degree of difficulty): {tautologies}, {DNF tautologies}, D3, {sub-graph pairs}.

Theorem-3

- For any $T_Q(k)$ of type **Q**, $\frac{T_Q(k)}{\frac{\sqrt{k}}{(\log k)^2}}$ is unbounded
- There is a $T_Q(k)$ of type **Q** such that $T_Q(k) \leqslant 2^{k(\log k)^2}$

Theorem-4

If the set S of strings is accepted by a non-deterministic machine within time $T(n) = 2^n$, and if $T_Q(k)$ is an honest (i.e. real-time countable) function of type Q, then there is a constant K, so S can be recognized by a deterministic machine within time $T_Q(K8^n)$.

First, he emphasized the significance of polynomial time reducibility. It means that if we have a polynomial time reduction from one problem to another, this ensures that any polynomial time algorithm from the second problem can be converted into a corresponding polynomial time algorithm for the first problem.

Second, he focused attention on the class NP of decision problems that can be solved in polynomial time by a non-deterministic computer. Most of the intractable problems belong to this class, NP.

Third, he proved that one particular problem in NP has the property that every other problem in NP can be polynomially reduced to it. If the satisfiability problem can be solved with a polynomial time algorithm, then every problem in NP can also be solved in polynomial time. If any problem in NP is intractable, then satisfiability problem must be intractable. Thus, satisfiability problem is the hardest problem in NP.

Fourth, Cook suggested that other problems in NP might share with the satisfiability problem this property of being the hardest member of NP.

2-Satisfiability (2-SAT) Problem

Boolean Satisfiability Problem

Boolean Satisfiability or simply **SAT** is the problem of determining if a Boolean formula is satisfiable or unsatisfiable.

- **Satisfiable :** If the Boolean variables can be assigned values such that the formula turns out to be TRUE, then we say that the formula is satisfiable.
- **Unsatisfiable :** If it is not possible to assign such values, then we say that the formula is unsatisfiable.

Examples:

- $F = A \wedge \bar{B}$, is satisfiable, because A = TRUE and B = FALSE makes F = TRUE.
- $G = A \wedge \bar{A}$, is unsatisfiable, because:

A	$\bar{A}$	G
TRUE	FALSE	FALSE
FALSE	TRUE	FALSE

Note : Boolean satisfiability problem is NP-complete (For proof, refer lank">Cook's Theorem)

What is 2-SAT Problem

```
2-SAT is a special case of Boolean Satisfiability Problem and can be so
in polynomial time.
```

To understand this better, first let us see what is Conjunctive Normal Form (CNF) or also known as Product of Sums (POS).

CNF : CNF is a conjunction (AND) of clauses, where every clause is a disjunction (OR).

Now, 2-SAT limits the problem of SAT to only those Boolean formula which are expressed as a CNF with every clause having only **2 terms**(also called **2-CNF**).

Example:

$$F = (A_1 \vee B_1) \wedge (A_2 \vee B_2) \wedge (A_3 \vee B_3) \wedge ..... \wedge (A_m \vee B_m)$$

Thus, Problem of 2-Satisfiabilty can be stated as:

Given CNF with each clause having only 2 terms, is it possible to assign such values to the variables so that the CNF is TRUE?

Examples:

Input : $F = (x1 \vee x2) \wedge (x2 \vee \bar{x1}) \wedge (\bar{x1} \vee \bar{x2})$
Output : The given expression is satisfiable.
(for x1 = FALSE, x2 = TRUE)

Input : $F = (x1 \vee x2) \wedge (x2 \vee \bar{x1}) \wedge (x1 \vee \bar{x2}) \wedge (\bar{x1} \vee$
Output : The given expression is unsatisfiable.
(for all possible combinations of x1 and x2)

Halting Problem: **It is an example of NP-Hard decision problem that is not NP-Complete.**

The halting problem A and an input I whether algorithm A with input I ever terminates. This problem un decidable. That is there exist no algorithm to solve this problem.

So it clearly can not be in NP.

DEFINITION :

- Let a, b and n are integers and n > 0.

  We write a ≡ b mod n if and only if n divides a − b.

  n is called the modulus.

  b is called the remainder.

For Example:

29 ≡ 15 mod 7 because 7|(29 − 15)

12 ≡ 3 mod 9 ; 3 is a valid remainder since 9 divides 12 − 3

12 ≡ 21 mod 9 ; 21 is a valid remainder since 9 divides 12 − 21

12 ≡ −6 mod 9 ; −6 is a valid remainder since 9 divides −6 − 3

1. $a \equiv b \pmod n$ *if* $n|(a-b)$
2. $a \equiv b \pmod n$ *implies* $b \equiv a \pmod n$
3. $a \equiv b \pmod n$ *and* $b \equiv c \pmod n$ **imply** $a \equiv c \pmod n$

Proof of 1.

If $n|(a-b)$, then $(a-b) = kn$ for some k. Thus, we can write

$a = b + kn$. Therefore,

$(a \bmod n)$ = (remainder when $b + kn$ is divided by n) = (remainder when b is divided by n) $(b \bmod n)$.

$23 \equiv 8 \pmod 5$ because $23 - 8 = 15 = 5 \times 3$

$-11 \equiv 5 \pmod 8$ because $-11 - 5 = -16 = 8 \times (-2)$

$81 \equiv 0 \pmod{27}$ because $81 - 0 = 81 = 27 \times 3$

TYPES OF MODULAR ARITHMETIC:

- We can add and subtract congruent elements without losing congruence:

$$[(a \bmod n) + (b \bmod n)] \bmod n = (a + b) \bmod n$$

$$[(a \bmod n) - (b \bmod n)] \bmod n = (a - b) \bmod n$$

- Multiplication also works:

$$[(a \bmod n) \times (b \bmod n)] \bmod n = (a \times b) \bmod n$$

Proof of 1.

Let $(a \bmod n) = Ra$ and $(b \bmod n) = Rb$. Then, we can write

$a = Ra + jn$ for some integer j and $b = Rb + kn$ for some integer k.

$$(a + b) \bmod n = (Ra + jn + Rb + kn) \bmod n$$

$$= [Ra + Rb + (k + j)\, n] \bmod n$$

$$= (Ra + Rb) \bmod n$$

$$= [(a \bmod n) + (b \bmod n)] \bmod n$$

EXAMPLES :

- $[(a \bmod n) + (b \bmod n)] \bmod n = (a + b) \bmod n$

$11 \bmod 8 = 3;\ 15 \bmod 8 = 7$

➡ $[(11 \bmod 8) + (15 \bmod 8)] \bmod 8 = 10 \bmod 8 = 2$

$(11 + 15) \bmod 8 = 26 \bmod 8 = 2$

- $[(a \bmod n) - (b \bmod n)] \bmod n = (a - b) \bmod n$

➡ $[(11 \bmod 8) - (15 \bmod 8)] \bmod 8 = -4 \bmod 8 = 4$

$(11 - 15) \bmod 8 = -4 \bmod 8 = 4$

- $[(a \bmod n) \times (b \bmod n)] \bmod n = (a \times b) \bmod n$

➡ $[(11 \bmod 8) \times (15 \bmod 8)] \bmod 8 = 21 \bmod 8 = 5$

$(11 \times 15) \bmod 8 = 165 \bmod 8 = 5$

Property	Expression
Cummitative Laws	$(w + x) \bmod n = (x + w) \bmod n$ $(w \times x) \bmod n = (x \times w) \bmod n$
Associative Laws	$[(w + x) + y] \bmod n = [w + (x + y)] \bmod n$ $[(w \times x) \times y] \bmod n = [w \times (x \times y)] \bmod n$
Distributive Law	$[w \times (x + y)] \bmod n = [(w \times x) + (w \times y)] \bmod n$
Identities	$(0 + w) \bmod n = w \bmod n$ $(1 \times w) \bmod n = w \bmod n$
Additive Inverse (-w)	For each $w^- Z_n$ there exists a z such that $w + z \equiv 0 \bmod n$