UNIT-V

DETERMINISTIC ALGORITHMS:

The algorithms, in which the result (or, outcome) of every operation is uniquely defined, are called *deterministic algorithms*.

NON-DETERMINISTIC ALGORITHMS:

- The algorithms, in which the outcomes of certain operations may not be uniquely defined but are limited to the specified sets of possibilities (i.e., possible outcomes), are said to be *non-deterministic algorithms*.
- The theoretical (or, hypothetical) machine executing such operations is allowed to choose any one of these possible outcomes.

- The non-deterministic algorithm is a two-stage algorithm.

  1. <u>Non-deterministic stage (or, Guessing stage):</u>
  Generate an arbitrary string that can be thought of as a candidate solution to the problem.

  2. <u>Deterministic stage (or, Verification stage):</u>
  This stage takes the candidate solution and the problem instance as input and returns "yes" if the candidate solution represents actual solution.

- To specify non-deterministic algorithms, we use three functions:
  1. *Choice(S)*: arbitrarily chooses one of the elements of set 'S'.
  2. *Success( )*: signals a successful completion.
  3. *Failure( )*: signals an unsuccessful completion.

- The assignment statement x: = Choice(1, n) could result in x being assigned with any one of the integers in the range [1, n].

  There is no rule specifying how this choice is to be made. That's why the name *non-deterministic* came into picture.

- The Failure( ) and Success( ) signals are used to define a completion of the algorithm.

- Whenever there is a particular choice (or, set of choices (or) sequence of choices) that leads to a successful completion of the algorithm, then that choice (or, set of choices) is always made and the algorithm terminates successfully.

- *A nondeterministic algorithm terminates unsuccessfully, if and only if* ***there exists no set of choices*** *leading to a success signal.*

- The computing times for **Choice( ), Failure( ), Success( )** are taken to be O(1), i.e., constant time.

- A machine capable of executing a non-deterministic algorithm is called *non-deterministic machine*.

EXAMLE: 1: NON-DETERMINISTIC SEARCH:

```
Algorithm Nsearch(A, n, x)
{
    //A[1:n] is a set of elements, from which we have to determine
    //an index j, such that A[j]:=x, or 0 if x is not present in A.

    // Guessing Stage
    j := Choice(l, n);

    // Verification Stage
    if A[j] = x then
    {
        write(j);
        Success( );
    }
    write(0);
    Failure( );

}
```

➔ The time complexity is O(1)

EXAMLE 2: NON-DETERMINISTIC SORTING:

Algorithm NSort(A, n)
// A[1:n] is an array that stores n elements, which are positive //integers.
// B[1:n] is an auxiliary array, in which elements are put at //appropriate positions. That means, B stores the sorted elements.

```
{
    // guessing stage
    for i := 1 to n do
    {
        j := Choice(l, n);      //guessing the position of A[i] in B
        B[j] := A[i];           //place A[i] in B[j]
    }
    // verification stage
    for i:= 1 to n -1 do
    {
        if (B[i] > B[i+ 1]) then      // if not in sorted order.
            Failure( );
    }
    Write(B[l : n]);            // print sorted list.
    Success( );
}
```

Time complexity of the above algorithm is O(n).

→ We mainly focus on *nondeterministic decision algorithms.*
Such algorithms produce either **'1'** or **'0'** (or, **Yes/No**) as their output.

→In these algorithms, a successful completion is made iff the output is 1. And, a 0 is output, iff there is no choice (or, sequence of choices) available leading to a successful completion.

→The output statement is implicit in the signals **Success( )** and **Failure( )**. No explicit output statements are permitted in a decision algorithm.

EXAMPLE: 0/1 KNAPSACK DECISION PROBLEM:

The knapsack decision problem is to determine if there is an assignment of 0/1 values to x_i, $1 \leq i \leq n$ such that $\sum_{i=1}^{n} p_i x_i \geq r$ and $\sum_{i=1}^{n} w_i x_i \leq M$. r is a given number. The p_i's and w_i's nonnegative numbers.

```
Algorithm DKP(p, w, n, M, r, x)
{
    W:= 0; P:= 0;
    for i := 1 to n do
    {
        x[i]:= Choice(0, 1);
        W := W + x[i] * w[i];
        P:=P+ x[i] * p[i];
    }
    if ((W>M) or (P < r)) then Failure( );
    else Success( );
}
```

THE CLASSES P, NP, NP-HARD AND NP-COMPLETE:

→**P** is the set of all decision problems solvable by a deterministic algorithm in polynomial time.

→ An algorithm *A* is said to have *polynomial complexity (or, polynomial time complexity)* if there exists a polynomial *p( )* such that the computing time of *A* is *O(p(n))* for every input of size *n*.

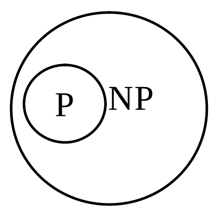
NP ***(Nondeterministic Polynomial time):***

→NP is the set of all decision problems solvable by a nondeterministic algorithm in polynomial time.

→A non-deterministic machine can do everything that a deterministic machine can do and even more. This means that all problems in class P are also in class NP. So, we conclude that P $\subseteq$NP.

→What we do not know, and perhaps what has become the most *famous unsolved problem* in computer science is, *whether $P = NP$ or $P \neq NP$*.

→The following figure displays the relationship between *P* and *NP* assuming that $P \neq NP$.



Some example problems in NP:

1. Satisfiability (SAT) Problem:

→SAT problem takes a Boolean formula as input, and asks whether there is an assignment of Boolean values (or, truth values) to the variables so that the formula evaluates to TRUE.

→A Boolean formula is a parenthesized expression that is formed from Boolean variables and Boolean operators such as OR, AND, NOT, IMPLIES, IF-AND-ONLY-IF.

→A Boolean formula is said to be in CNF (Conjunctive Normal Form, i.e., Product of Sums form) if it is formed as a collection of sub expressions called clauses that are combined using AND, with each clause formed as the OR of Boolean literals. A *literal* is either a variable or its negation.

→The following Boolean formula is in CNF:

$(x_1 \vee x_3 \vee x_5 \vee x_7) \wedge (x_3 \vee x_5) \wedge (x_6 \vee x_7)$

→The following formula is in DNF (Sum of Products form):

$(x_1 \wedge x_2 \wedge x_5) \vee (x_3 \wedge x_4) \vee (x_5 \wedge x_6)$

→ *CNF-SAT* is the SAT problem for CNF formulas.

→It is easy to show that *SAT* is in *NP*, because, given a Boolean formula $E(x_1, x_2, \ldots, x_n)$, we can construct a polynomial time non-deterministic algorithm that could proceed by simply choosing (nondeterministically) one of the 2^n possible assignments of truth values to the variables $(x_1, x_2, \ldots, x_n)$ and verifying that the formula $E(x_1, x_2, \ldots, x_n)$ is **true** for that assignment..

Nondeterministic Algorithm for *SAT* problem:

```
Algorithm NSAT(E, n)
{
    //Determine whether the propositional formula E is satisfiable.
    //The variables are x1, x2, …, xn.

    // guessing stage.
    for i:=1 to n do          // Choose a truth value assignment.
        xi := Choice(false, true);

    // verification stage.
     if E(x1, x2, …, xn) = true then Success( );
    else Failure( );
}
```

→Time complexity is O(n), which is a polynomial time. So, *SAT* is NP problem.

2. CLIQUE PROBLEM:

Clique: A *clique* of a graph 'G' is a complete subgraph of G.
→The size of the clique is the number of vertices in it.

Clique problem: Clique problem takes a graph 'G' and an integer 'k' as input, and asks whether G has a clique of size at least 'k'.

Nondeterministic Algorithm for Clique Problem:

Algorithm DCK(G, *n*, *k*)
{
    //The algorithm begins by trying to form a set of k distinct
    //vertices. Then it tests to see whether these vertices form a
    //complete sub graph.

    // guessing stage.
    S := Ø;   *// S is an initially empty set.*
    for *i* := 1 **to** *k* **do**
    {
        t := **Choice**(l, n);
        S := S U {t}   *// Add t to set S.*
    }
    //At this point, S contains k distinct vertex indices.
    //Verification stage
    for all pairs *(i, j)* such that i ∈ S, j ∈ S, and i ≠ j **do**
        if (i, j) is not an edge of G **then Failure( )**;
    Success( );
}

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

→**<u>Tractable Problems:</u>** Problems that can be solved in polynomial time are called ***tractable***.
<u>Intractable Problems:</u> 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.

<u>REDUCIBILITY:</u>

→ 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 *<u>Yes</u>* instances of D_1 to *<u>Yes</u>* instances of D_2 and all *<u>No</u>* instances of D_1 to *<u>No</u>* instances of D_2.

2. t is computable by a polynomial time algorithm.

→The definition for $\underline{D_1 \propto D_2}$ 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 $D_1 \propto D_3$.

<u>NP-HARD CLASS:</u>

→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'.

<u>Ex:</u> 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:

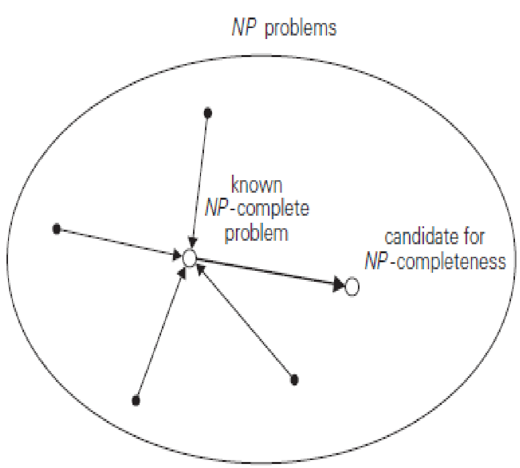
→A problem 'L' is said to be NP-Complete if 'L' is NP-Hard and L ∈ *NP*.

→These are the hardest problems in NP set.

Ex: SAT and Clique problems.

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

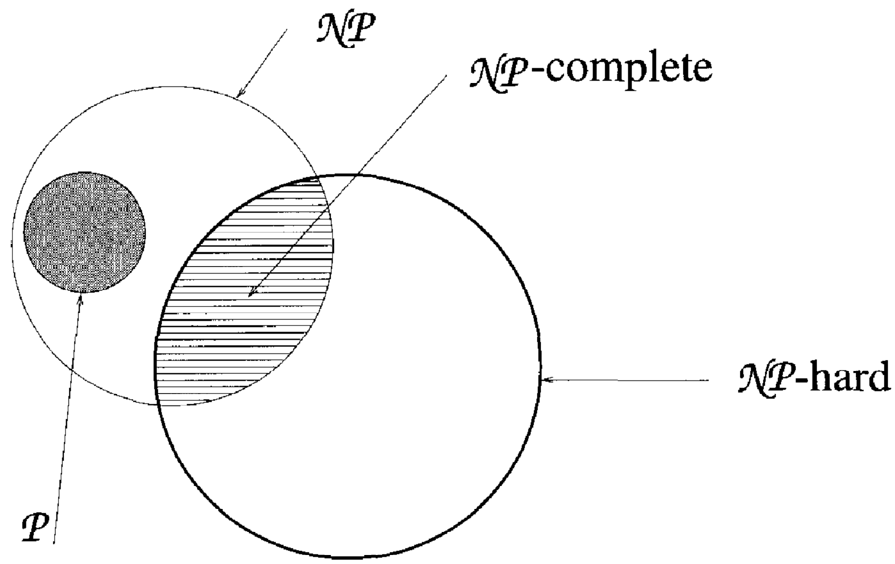
It can be done in two steps:
Step1:
Show that the problem in question is in *NP*; i.e., a randomly generated string can be checked in polynomial time to determine whether or not it represents a solution to the problem. Typically, this step is easy.
Step2:
Show that the problem in question is NP-Hard 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
