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

  2. Deterministic stage (or, Verification stage):
     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*.

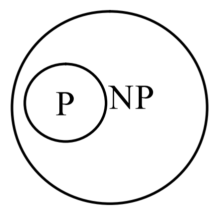
NP ***(Nondeterministic Polynomial time):***

→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 ⊆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 ≠ NP*.

→The following figure displays the relationship between *P* and *NP* assuming that *P ≠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 $E(x_1, x_2, ..., x_n)$, 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 $(x_1, x_2, ..., x_n)$ and verifying that the formula $E(x_1, x_2, ..., x_n)$ 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( );
}
```