NP Problems & Approximation Algorithms:-

NP Problems:- Complexity class - P, NP, NP complete, NP Hard. Reducibility, Cook's theorem.

Approximation Algorithms:- Introduction, Absolute Approximation, ε-approximation, polynomial time Approximation.

NP Problems:-

Complexity classes:-

Definition of NP class problem:- The set of all decision-based-problem come into division of NP problems. Who can't be solved or produced an output within polynomial time but verified in the polynomial time. NP class contains P class as a subset. NP problems being hard to solve.

Definition of P class problem:- The set of decision-based problems come into the division of P problems who can be solved (or) produced an output within polynomial time. P problems being easy to solve.

Definition of NP-complete class:-

A problem is in NP-complete if

1. It is in NP
2. It is NP-Hard,

Definition of NP-Hard class:-

The following conditions are satisfied then ~~it~~ it is said to be NP-Hard class problem.

1. If we solve this problem in polynomial time, then we can solve all NP problems in polynomial time.
2. If we convert the issue into one form to another form with in the polynomial time.

NP-Hard and NP-Complete

polynomial time	Exponential time
linear search - n	0/1 knapsack - 2^n
Binary search - $\log n$	Travelling SP - 2^n
Insertion sort - n^2	Sum of sub sets - 2^n
mergesort - $n\log n$	Graphcoloring - 2^n
matrixmultiplication - n^3	Hamiltonian cycle - 2^n.

Our research is to solve exponential time algorithms in polynomial time ① when we are unable to get this then atleast we are ~~going to~~ trying to show that we are trying to show similarity between them. so that if one problem is solved then all the otherproblems also solved.

② If we are unable ~~find~~ write deterministic algorithms, then try to write non-deterministic algorithms.

Ex:- In deterministic algorithms we know how each and every stmt works.

```
Algorithm NSearch(A, n, key)
{
    j = choice();  → 1
    if (key = A[j])
    {
        write(j);  → 1
        success();
    }
    write(0);
    failure();  → 1
}
```

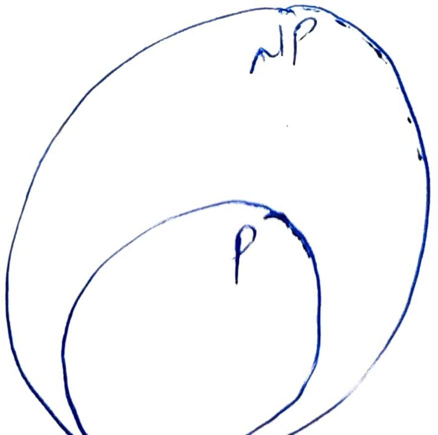
$\overline{O(1)}$

choice(); These will assumed
write(); to take 1 unit of
failure(); time.

choice() is giving us the index of the key element.
Here choice() how came to know that the key element is present at
the jth index? That's what it is non-deterministic. If we came to
know then it will be takes 1 unit of time to find an eleme
Here we don't know how choice is going to work so it's calle
as "magic" once we came to know we call it as "logic".

P class - These are deterministic algorithms, which takes polynom
time.

NP class - These are non deterministic but they take polynomial
time.



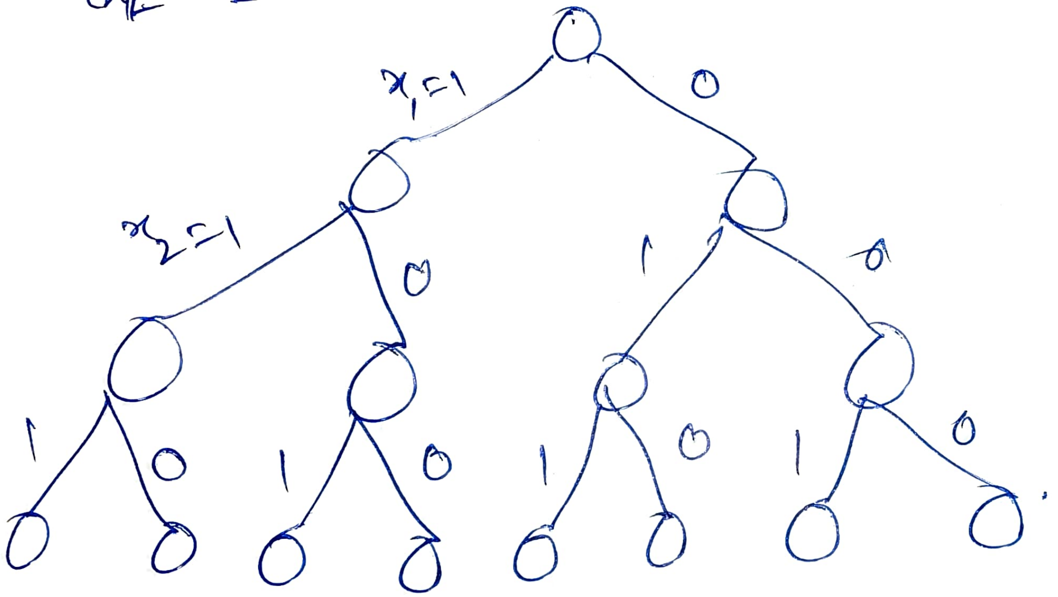
For relating exponential time algorithms we need some basic problem.

<u>Satisfiability Problem</u>:-

$x_i = \{x_1, x_2, x_3\}$

$CNF = (x_1 \vee \bar{x}_2 \vee \bar{x}_3 \vee x_4) \wedge (\bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_3)$

CNF =



If Exponential time problems are solved using above statespace tree they can be said to be deterministic algorithm

Cook–Levin theorem or Cook's theorem

In computational complexity theory, the Cook–Levin theorem, also known as Cook's theorem, states that the Boolean satisfiability problem is NP-complete. That is, it is in NP, and any problem in NP can be reduced in polynomial time by a deterministic Turing machine to the Boolean satisfiability problem.

*Stephen Arthur Cook and L.A. Levin in 1973 independently proved that the **satisfiability problem(SAT)** is NP-complete. Stephen Cook, in 1971, published an important paper titled 'The complexity of Theorem Proving Procedures', in which he outlined the way of obtaining the proof of an NP-complete problem by reducing it to **SAT**. He proved **Circuit-SAT** and **3CNF-SAT** problems are as hard as **SAT**. Similarly, Leonid Levin independently worked on this problem in the then Soviet Union. The proof that **SAT** is NP-complete was obtained due to the efforts of these two scientists. Later, Karp reduced 21 optimization problems, such as Hamiltonian tour, vertex cover, and clique, to the **SAT** and proved that those problems are NP-complete.*

<u>*Hence,* an ***SAT** is a significant problem and can be stated as follows:*</u>

Given a boolean expression **F** having **n** variables **$x_1,x_2,\ldots,x_n$**, and Boolean operators, is it possible to have an assignment for variables true or false such that binary expression **F** is true?

This problem is also known as the **formula – SAT**. An **SAT(formula-SAT or simply SAT)** takes a Boolean expression F and checks whether the given expression(or formula) is satisfiable. A Boolean expression is said to be satisfactory for some valid assignments of variables if the evaluation comes to be true. Prior to discussing the details of SAT, let us now discuss some important terminologies of a Boolean expression.

- **<u>Boolean variable:</u>** A variable, say **x**, that can have only two values, true or false, is called a boolean variable
- **<u>Literal:</u>** A literal can be a logical variable, say **x**, or the negation of it, that is **x** or **$\bar{x}$**; **x** is called a positive literal, and **$\bar{x}$** is called the negative literal
- **<u>Clause:</u>** A sequence of variables**($x_1,x_2,\ldots,x_n$)** that can be separated by a logical **OR** operator is called a clause. For example, **(x_1 V x_2 V x_3)** is a clause of three literals.
- **<u>Expressions:</u>** One can combine all the preceding clauses using a Boolean operator to form an expression.
- **<u>CNF form:</u>** An expression is in CNF form(conjunctive normal form) if the set of clauses are separated by an **AND (^),** operator, while the literals are connected by an **OR (v)** operator. The following is an example of an expression in the CNF form:
    - ***$f = (x_1 \vee \bar{x}_2 \vee x_3) \wedge (x_1 \vee \bar{x}_3 \vee x_2)$***
- **<u>3 – CNF:</u>** An expression is said to be in 3-CNF if it is in the conjunctive normal form, and every clause has exact three literals.

Thus, an **SAT** is one of the toughest problems, as there is no known algorithm other than the brute force approach. A brute force algorithm would be an exponential-time algorithm, as **2^n** possible assignments need to be tried to check whether the given Boolean expression is true or not. Stephen Cook and Leonid Levin proved that the **SAT** is NP-complete.

Cook demonstrated the reduction of other hard problems to SATs. Karp provided proof of 21 important problems, Such as Hamiltonian tour, vertex cover, and clique, by reducing it to SAT using Karp reduction.

Let us briefly introduce the three types of SATs. They are as follows:

Circuit- SAT: A circuit-SAT can be stated as follows: given a Boolean circuit, which is a collection of gates such as AND, OR, and NOT, and n inputs, is there any input assignments of Boolean variables so that the output of the given circuit is true? Again, the difficulty with these problems is that for n inputs to the circuit. 2n possible outputs should be checked. Therefore, this brute force algorithm is an exponential algorithm and hence this is a hard problem.

CNF-SAT: This problem is a restricted problem of SAT, where the expression should be in a conjunctive normal form. An expression is said to be in a conjunction form if all the clauses are connected by the Boolean AND operator. Like in case of a SAT, this is about assigning truth values to n variables such that the output of the expression is true.

3-CNF-SAT(3-SAT): This problem is another variant where the additional restriction is that the expression is in a conjunctive normal form and that every clause should contain exactly three literals. This problem is also about assigning n assignments of truth values to n variables of the Boolean expression such that the output of the expression is true. In simple words, given an expression in 3-CNF, a 3-SAT problem is to check whether the given expression is satisfiable.

These problems can be used to prove the NP-completeness of some important problems.