The **class P** consists of problems that can be solved in polynomial time, meaning their TC is bounded by a polynomial function of the input size. For example, if a problem can be solved in time O(n^2) or O(n^3), where n is the size of the input, then it belongs to class P.

The **class NP** (Non-deterministic Polynomial time) consists of problems for which a proposed solution can be verified in polynomial time. This means that if someone provides a solution to an NP problem along with a "certificate" that helps verify the correctness of the solution, the verification process itself can be completed in polynomial time.  
(Problems are ones where if someone hands you a possible solution along with a quick way to check if it's correct, then that problem belongs to NP. This "quick check" is like a certificate that confirms the solution's validity, and it can be verified in polynomial time.)

A **Hamiltonian cycle** in a graph is a cycle that visits every vertex exactly once and returns to the starting vertex. Determining whether a directed graph has a Hamiltonian cycle is a problem that, as of now, does not have a known polynomial time algo. However, if someone provides a sequence of vertices claiming it to be a Hamiltonian cycle, we can verify whether it indeed forms a cycle that visits each vertex exactly once in polynomial time by checking if it starts & ends at same vertex, & if each vertex appears exactly once in sequence.

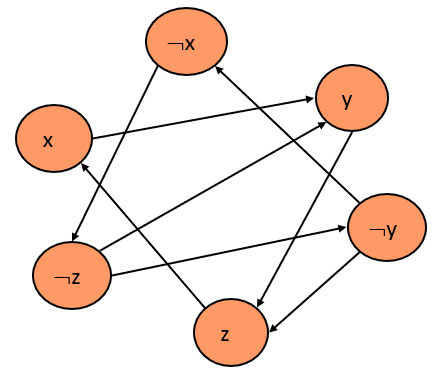
**SAT (Boolean Satisfiability Problem)** involves determining whether a given boolean formula can be satisfied, meaning there exists an assignment of truth values (0 or 1) to its variables that makes the entire formula evaluate true.

In **CNF (Conjunctive Normal Form)**, boolean formulas are represented as conjunction (AND operation, reverse V) of clauses, where each clause is disjunction (OR operation V) of literals (variables or negations of variables).



**2-CNF SAT:** This refers to a specific type of Boolean formula where each clause (part separated by 'AND') contains at most two literals (variables or their negations). For example, (x OR y) AND (NOT y OR z) is a 2-CNF formula.

To solve 2-CNF SAT, we convert each OR clause into an implication clause. For instance, if we have (x OR y), we transform it into its equivalent implication form: (NOT x => y) AND (NOT y => x) "if x is false, then y must be true" and "if y is false, then x must be true". This conversion maintains the logical equivalence of the formula.

Once we have the formula in implication form, we can represent it as a directed graph, where variables are nodes and implications are directed edges. Now, to check if the formula is satisfiable, we search for paths in this graph using algorithms like DFS or BFS.  
If during the path search, we find a path from a variable to its negation (for example, a path from x to NOT x), or from its negation to the variable, it implies a contradiction. In a satisfiable formula, such paths shouldn't exist because assigning both a variable and its negation to true at the same time would lead to a contradiction.



As for whether **3-SAT** is in NP, the answer is yes. Even though there's no known polynomial-time algorithm to solve it, if someone provides a solution, you can verify it in polynomial time by simply checking whether the given assignment satisfies each clause in the formula. This property makes 3-SAT an NP problem.

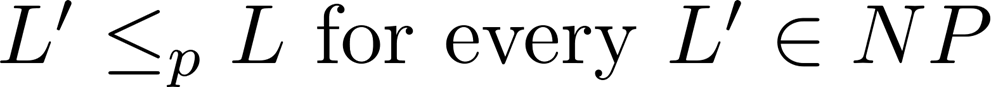
P is considered a subset of NP because problems in P can be efficiently verified. However, it's unknown whether NP is a subset of P, a question that remains one of the most famous unsolved problems in computer science known as the P vs NP problem.

1. *Undecidable Problems:* These are problems for which no algo can correctly solve every instance of the problem. Hilbert's nth problem is impossible to devise a general algo that can determine whether any given polynomial equation with integer coefficients has integer roots. These problems are beyond the scope of NP because they cannot even be efficiently verified in polynomial time, let alone solved.
2. *Tautology:* A tautology is a boolean formula that is always true, regardless of the truth values assigned to its variables. Verifying whether a given boolean formula is a tautology requires checking all possible truth assignments, which grows exponentially with the number of variables. While it's not feasible to verify a tautology in polynomial time, it's still within the realm of NP because, theoretically, if someone provided proof of a formula being a tautology, it could be verified in polynomial time.

**Reducibility**: to compare the computational complexity of different problems. It essentially asks whether one problem can be transformed into another in such a way that solving the second problem provides a solution to the first.  
For example, you can reduce a linear equation to a quadratic equation by setting the coefficient of the square term to 0. This transformation shows that solving the quadratic equation can solve the linear equation as well.

**NP-hardness** is a property of problems that are at least as hard as the hardest problems in NP. A problem is NP-hard if every problem in NP can be polynomial-time reduced to it.

A reduction from problem A to problem B means that an algorithm for solving problem B can be used to solve problem A. This reduction must be computable in polynomial time.  
If a problem L' is reducible in polynomial time to another problem L, and L is in NP, then L' is NP-hard. This means that solving L' is at least as hard as solving any problem in NP, because any problem in NP can be transformed into L' efficiently.



**NP-Complete Problems/Languages**: These are problems that satisfy two conditions:

1. They are in NP (solutions can be verified in polynomial time).
2. They are NP-hard (every problem in NP can be polynomial-time reduced to them).

*Reducibility* is transitive, meaning if problem A can be reduced to problem B, and problem B can be reduced to problem C, then problem A can also be reduced to problem C.

*To show that a problem P' is NP-complete*, you typically follow these steps:

1. Start with a known NP-complete problem P.
2. Show that P can be reduced to P' (in polynomial time).
3. Since all problems in NP can be reduced to P (because P is NP-complete), and P can be reduced to P', then all problems in NP can also be reduced to P'.

Two classic examples are SAT and 3-CNF SAT. Any boolean formula can be reduced to 3-CNF SAT form, which is why it's often used as a starting point for reductions.

***Reducing 3-CNF SAT to CLIQUE***:-

1. *3-CNF SAT:* This is a type of boolean formula where each clause contains at most three literals and the formula is in CNF. The problem is to determine if there exists an assignment of truth values to the variables that makes the entire formula true.
2. *CLIQUE:* This problem involves finding a clique in an undirected graph, where a clique is a subset of vertices such that every pair of vertices in the subset is connected by an edge. Problem asks if there is a clique of a certain size in graph.

Given: You start with a boolean formula in 3-CNF SAT form.

Goal: You want to construct a graph in such a way that the formula is satisfiable if and only if the graph contains a clique of a certain size.

1. *Constructing the Graph:* For each clause in the 3-CNF SAT formula, you create a gadget in the graph. This gadget represents all possible truth assignments that satisfy the clause. Each gadget will have vertices representing literals and edges representing conflicts between literals. The gadgets are designed such that if and only if the formula is satisfiable, there exists a clique of a certain size in the graph.
2. *Encoding Satisfiability:* The graph is constructed in such a way that there exists a clique of a certain size if and only if the boolean formula is satisfiable. In other words, if you find a clique of the specified size in the graph, you can translate it back to a truth assignment that satisfies the original boolean formula, and vice versa.

**Decision Problems vs. Optimization Problems:** NP-completeness is typically discussed in the context of decision problems, where the answer is simply "yes" or "no". Even problems that are naturally formulated as optimization problems (like finding the maximum clique size) can be transformed into decision problems.

INCOMPLETE  
I AIN’T UNDERSTANDING SHIT.