# NP-Complete

## 1 Definition

In computer science, there exist some problems whose solutions are not yet found, the problems are divided into classes known as **Complexity Classes**. In complexity theory, a Complexity Class is a set of problems with related complexity. These classes help scientists to group problems based on how much time and space they require to solve problems and verify the solutions. It is the branch of the theory of computation that deals with the resources required to solve a problem. There are multiple types of complexity classes:

### 1.1 P Class

The P in P class stands for **Polynomial Time**, it is the collection of decision problems (problems with a "yes" pr "no" answer) that can be solved by a deterministic machine in polynomial time ($O(n^d)$). Notice that $O(\log n)$ is also polynomial time, based on the definition.

The solution to P problems is easy to find. P is often a class of computational problems that are **solvable and tractable**. Tractable means that the problems could be solved **in theory as well as in practice**. The problems could be solved in theory but not in practice are known as **intractable**.

### 1.2 NP Class

The NP in NP class stands for **Non-deterministic Polynomial Time**. It's the collection of decision problems that can be solved by a **non-deterministic machine** in polynomial time. The solutions of the NP class are hard to find, **but the solutions are easy to verify**. Problems of NP could be verified by a **Turing machine** in polynomial time. Notice that **NP includes P**!

#### 1.2.1 Non-Deterministic Machine

Based on the definition, a non-deterministic machine could be in **multiple states** at once and can make arbitrary choices between different computational paths. In other words, at any point in its computation, can branch into many possible futures. Each state can have multiple possible transitions for a given input symbol.

The **computation** of a non-deterministic machine can be visualized as a tree, where each path from the root to a leaf represents a possible sequence of choices and transitions. **If at least one of these paths leads to an accepting state (for decision problems), the machine accepts the input.**

#### 1.2.2 Turing Machine

A **deterministic** Turing machine (DTM) is a theoretical model of computation where, given the current state and the symbol being read from the tape, the machine has a single, uniquely defined transition to a next state, a symbol to write, and a direction to move the tape head (left or right).

#### 1.2.3 Examples

Let us consider an example to better understand the NP class. Suppose there is a company having a total of 1000 employees having unique employee IDs. Assume that there are 200 rooms available for them. A selection of 200 employees must be paired together, but the CEO of the company has the data of some employees who can't work in the same room due to personal reasons.

This is an example of an NP problem. Since it is easy to check if the given choice of 200 employees proposed by a coworker is satisfactory or not i.e. no pair taken from the coworker list appears on the list given by the CEO. But generating such a list from scratch seems to be so hard as to be completely impractical.

### 1.3 NP-Hard Class

An NP-hard problem is **at least as hard as the hardest problem** in NP and it is a class of problems such that **every problem in NP could reduce to NP-hard**. Some features include:

- Not all NP-hard problems are in NP
- If a solution for an NP-hard problem is given then it takes a long time to check whether it is right or not
- A problem $A$ is in NP-hard if **for every problem $L$ in NP, there exits a polynomial time reduction from $L$ to $A$**

### 1.4 NP-Complete Class

A problem is NP-complete if it is both NP and NP-hard. NP-complete problems are the hard problems in NP. Some features include:

- NP-complete problems are special, **any problem in NP could be transformed or reduced into NP-complete problems in polynomial time**.
- If one could solve an NP-complete problem in polynomial time, then **one could also solve any NP problem in polynomial time.**

### 1.5 Interrelations

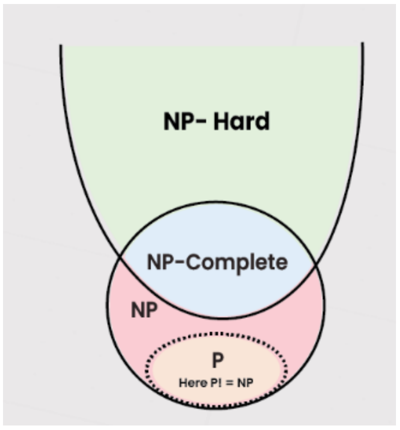

Figure 1: Complexity Classes Interrelations

Or, it could be possible that:

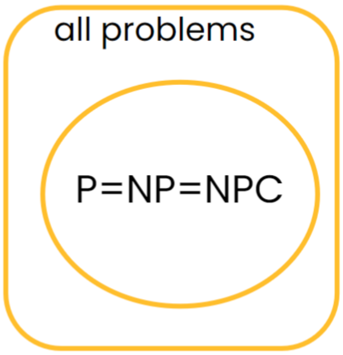

Figure 2: Possible NP relation

## 2 Reductions

### 2.1 Definition

A reduction from $A$ to $B$ is showing that **we can solve $A$ using the algorithm that solves $B$**, or we say **problem $A$ is easier than problem $B$**:

$$A \leq B \tag{1}$$

Given two problems $A$, $B$, we say that $A$ is **polynomially reducible** to $B$ ($A \leq_p B$)if:

- Problem $A$ can be reduced to problem $B$ if there exist a function $f$ that transforms instances of $A$ into instances of $B$ such that $i$ is a YES instance of $A$ **if and only if** $f(i)$ is a YES instance of $B$
- The function $f$ must be computable in polynomial time

### 2.2 Implication

There are several implications of polynomial-time reductions:

- **Design algorithms:** if $X \leq_p Y$ and $Y$ could be solved in polynomial time (PT), then $X$ could also be solved in PT.
- **Establish intractability:** if $X \leq_p Y$ and $X$ cannot be solved in PT, then $Y$ can not be solved in PT.
- **Establish equivalence:** if $X \leq_p Y$ and if $Y \leq_p X$, we can have $X =_p Y$
- **Transitivity:** if $X \leq_p Y$ and $Y \leq_p Z$, then $X \leq_p Z$

## 3 Recipe

Recipe to establish **NP-completeness** of problem $Y$:

1. **Step 1:** Show that $Y$ is in NP:
   - Describe how a potential solution will be represented
   - Describe a procedure to check whether the potential solution is a correct solution to the problem instance, and argue that this procedure takes **polynomial time**.
2. **Step 2**: Choose an NP-complete problem $X$.
3. **Step 3**: Prove that $X \leq_P Y$ ($X$ is poly-time reducible to $Y$):
   - Describe a procedure $f$ that converts the inputs $i$ of $X$ to inputs of $Y$ in polynomial time.
   - Show that the reduction is correct by showing that $X(i) = YES$ if and only if $Y(f(i)) = YES$
   - The reason why we are **reducing an existing NP-Complete problem to the new problem** is that: this means $X$ is easier than $Y$, so $Y$ is **at least as hard as NP-Complete**, so $Y$ is also NP-Complete.
4. **Step 4**: Justification: If $X$ is an NP-complete problem, and $Y$ is a problem in NP with the property that $X \leq_P Y$, then $Y$ is NP-complete.

The hardest part is step 3. The recipe for this step is shown below:

- Let $I_1$ be any instance of $X$.
- Transform $I_1$ into an instance $I_2$ of problem $Y$.
- Check whether this transformation takes a polynomial time.
- **Suppose $I_1$ has a solution**, then prove that $I_2$ also has a solution.
- **Suppose $I_2$ has a solution**, then show that it implies that $I_1$ has a solution.

# 4 Classical NP-complete Problems

## 4.1 Boolean Satisfiability Problem (SAT)

### 4.1.1 Definition

SAT is the **first known NP-complete problem** and laid the foundation for the theory of NP-completeness. Before explaining the definition of SAT, we first need to introduce **Conjunctive Normal Form (CNF)**.

CNF is a way of structuring a Boolean formula such that it is expressed as an AND of ORs. We call AND as **conjunction**, and OR as **disjunction**. Here is an example of CNF:

$$(A \vee \neg B) \wedge (\neg A \vee B \vee C) \wedge (\neg C \vee D) \tag{2}$$

Some important concepts:

1. **Literal**: a variable or the negation of a variable. In Boolean logic, literals are the basic building blocks of clauses. For example, a **positive literal** is $A$, and the **negative literal** is $\neg A$
2. **Clause**: a **disjunction (only !!!)** of literals, such as $(\neg A \vee B \vee C)$. The **conjunctions (only !!!)** of clauses will be CNF.
3. **AND ($\wedge$)**: the result is true if both operands are true.
4. **OR ($\vee$)**: The result is true if at least one of the operands is true.
5. **NOT ($\neg$)**: The result is true if the operand is false, and vice versa.

Now, the definition of SAT will be:

- **Input:** A Boolean formula in CNF, where each clause can have any number of literals.
- **Problem:** Determine if there is an assignment of **truth values** to the variables that makes the entire formula true. This is also the definition of **satisfiable**.

### 4.1.2 Prove NP-Completeness

Recall the recipe, to prove NP-Completeness of a problem, we first need to prove it is NP, then prove that a NP-Complete problem is polynomially reducible to this problem. However, **this problem is the first NP-Complete problem**, so we need to prove that it is also **NP-Hard**, to prove that it is **NP-Complete**.

1. **SAT is in NP:** A problem is in NP if a given solution can be verified in polynomial time. The **verification** procedure is to evaluate each clause to check if **at least one literal** in the clause is true under the given assignment. Because the verification process could be done in **linear time** relative to the number of clauses and the number of literals in each clause. Therefore, SAT is in NP.

2. **SAT is NP-Hard**: To show that SAT is NP-hard, we need to demonstrate that any problem in NP could be **reduced to SAT in polynomial time.** This is achieved by **Cook-Levin Theorem**, Here is a brief procedure of this:
   - Take any problem in NP, which can be solved by a nondeterministic Turing machine (NTM) in polynomial time.
   - Construct a Boolean formula that is satisfiable if and only if the NTM accepts the input.

By doing this, we can now prove that SAT problem is NP-Complete.

## 4.2 3-SAT Problem

### 4.2.1 Definition

With the definition of SAT, 3-SAT problem could be simplified:

- **Input:** A Boolean formula in 3-CNF, where each clause has **exactly three literals**.
- **Problem:** Determine if there is an assignment of truth values to the variables that make the entire formula true.

### 4.2.2 Prove NP-Completeness

Recall the recipe, the proof includes:

1. **3-SAT is in NP**: check each clause to see if at least one literal in the clause is true under the given assignment.
2. **Reduction from NP-Complete:** In SAT, there are random number of literals in one clause, so we need to transfer it into exactly three literals. There will be two cases:
   - **Clauses with fewer than three literals:** If a clause has **only one literal (**$A$**)**, then we need to add two dummy variables $x$ and $y$:

     $$A = (A \vee x \vee y) \wedge (A \vee x \vee \neg y) \wedge (A \vee \neg x \vee y) \wedge (A \vee \neg x \vee \neg y) \tag{3}$$

     With this construction, regardless of the values of $x$ and $y$, $A$ being true will satisfy all four clauses. When $A$ is false, the choice of $x$ and $y$ will still have the ability to control the result, either TRUE or FALSE. Therefore, this construction is valid, and will preserves satisfiability. Similarly, if a clause has **two literals**, then we add one dummy variable $x$:

     $$(A \vee B) = (A \vee B \vee x) \wedge (A \vee B \vee \neg x) \tag{4}$$

- **Clauses with more than three literals:** If a clause has more than three literals, then we can break it down into multiple clauses. For example, assume we have a clause:

  $$Cr = (A \vee B \vee C \vee D \vee \cdots) = (A \vee B \vee EXT) \tag{5}$$

  We can introduce a new variable $x$ to get the new clause:

  $$Cr' = (A \vee B \vee x) \wedge (\neg x \vee EXT) \tag{6}$$

  The same procedure can be applied repeatedly to $Cr'$ until there are no more clauses with more than three literals remaining.

Both of these constructions will take linear steps with the number of literals in one clause and the number of clauses, so the reduction is in polynomial time.

3. **Prove Correctness**: assume there is an instance $i$, and the reduction process as $f$, we need to prove SAT$(i)$ = YES (satisfiable) if and only if 3-SAT$(f(i))$ = YES (satisfiable).

   - **If $Cr$ satisfiable, then $Cr'$ satisfiable.** This means:

     $$Cr = (A \vee B \vee EXT) = 1 \tag{7}$$

     And we want:

     $$Cr' = (A \vee B \vee x) \wedge (\neg x \vee EXT) = 1 \tag{8}$$

     If either $A = 1$ or $B = 1$, then we assign $x = 0$:

     $$Cr' = (A \vee B \vee 0) \wedge (1 \vee EXT) = (A \vee B) \wedge (1) = 1 \tag{9}$$

     If $A = 0$ and $B = 0$, we can assign $x = 1$ (now $EXT$ must be 1 to satisfy $Cr$):

     $$Cr' = (A \vee B \vee 1) \wedge (0 \vee EXT) = (1) \wedge (EXT) = 1 \tag{10}$$

   - **If $Cr'$ satisfiable, then $Cr$ satisfiable.** This means:

     $$(A \vee B \vee x) = 1, \;\; (\neg x \vee EXT) = 1 \tag{11}$$

     And we want:

     $$Cr = (A \vee B \vee EXT) = 1 \tag{12}$$

     If $x = 0$, then:

     $$Cr' = (A \vee B \vee 0) \wedge (1 \vee EXT) = (A \vee B) \wedge (1) = 1 \tag{13}$$

     therefore:

$$(A \vee B) = 1,\ \ Cr = 1 \tag{14}$$

If $x = 1$, then:

$$Cr' = (A \vee B \vee 1) \wedge (0 \vee EXT) = (1) \wedge (EXT) = 1 \tag{15}$$

therefore:

$$(EXT) = 1, Cr = 1 \tag{16}$$

Therefore, we can prove that reduction is valid.

4. **Justification:** Because SAT is an NP-Complete problem, 3-SAT is a problem in NP, and SAT is polynomial reducible to 3-SAT, therefore we can prove that 3-SAT is NP-Complete.

## 4.3 Vertex Cover Problem (VC)

Vertex cover is a classic problem in graph theory:

- **Input:** A graph $G = (V, E)$ and an integer $k$
- **Problem:** Determine whether there exists a subset $V' \subseteq V$ of size at most $k$ such that every edge in $E$ is incident (connected) to at least one vertex in $V'$. In other words, we need to find a set of vertices such that **every edge in the graph has at least one endpoint in this set**, and the size of this set is as small as possible.

## 4.4 Traveling Salesman Problem (TSP)

- **Inputs:**
  - A set of cities.
  - A distance matrix or a function that gives the distance between each pair of cities.
- **Problem:** Given a list of cities, distances between each pair of cities, and a number $D$, determine if there exists a route that visits each city exactly once, returns to the origin city, and has a total distance less than or equal to $D$.

## 4.5 Knapsack Problem

- **Inputs:**
  - A set of items $X$, nonnegative weights $w_i$, nonnegative values $v_i$.
  - A weight limit $W$, and a target value $V$.

- **Problem:** Is there a subset $S \subseteq X$ such that:

$$\sum_{i \in S} w_i \leq W \tag{17}$$

$$\sum_{i \in S} v_i \geq V \tag{18}$$

### 4.6 Hamiltonian Cycle Problem

- **Input:** A graph $G = (V, E)$, where $V$ is the set of vertices and $E$ is the set of edges.
- **Problem**: Determine if there exists a cycle that visits each vertex exactly once and returns to the starting vertex.
- **Features:**
  - It is different from TSP. The goal is to find a cycle that includes every vertex exactly once.
  - It is a decision problem, which means the answer is simply ”yes” or ”no”.
  - It does not necessarily involve costs or distances on the edges.

### 4.7 Clique Problem

- **Inputs:**
  - A graph $G = (V, E)$
  - A positive integer $k$
- **Problem:** Determine if there exists a subset of vertices $V' \subseteq V$ such that every pair of vertices in $V'$ is connected by an edge, and the size of $V'$ is at least $k$.

### 4.8 Graph Coloring Problem

- **Inputs**:
  - A graph $G = (V, E)$
  - A number of colors $k$
- **Problem**: Determine if it is possible to color the vertices of the graph using at most $k$ colors such that no two adjacent vertices have the same color.

## 4.9 Independent Set Problem

- **Inputs**:
  - A graph $G = (V, E)$, where $V$ is the set of vertices and $E$ is the set of edges
  - A positive integer $k$
- **Problem**: Determine if there exists an independent set of size at least $k$. An independent set is a subset of vertices $I \subseteq V$ such that no two vertices in $I$ are adjacent. (There is no edge between any pair of vertices in $I$)

## 4.10 Set Cover Problem

- **Inputs:**
  - A set $U$ of elements
  - A collection $S_1, S_2, \cdots, S_m$ of subsets of $U$
  - An integer $k$
- **Problem:** Does there exist a collection of at most $k$ of these sets whose union is equal to $U$?