NP-Completeness

1. Define P, NP, NP-Complete, NP-Hard complexity classes.

P (Polynomial Time):

The set of **decision problems** that can be **solved** by a deterministic Turing machine in **polynomial time**.

Informal: Problems that can be solved "quickly" (in time (O(n^k)) for some constant (k)).

Example problems:

- Sorting, shortest path in a graph, minimum spanning tree.
- Determining if a number is prime (AKS primality test, 2002).

NP (Nondeterministic Polynomial Time):

The set of **decision problems** for which a given "yes" solution can be **verified** in **polynomial time** by a deterministic Turing machine, given a suitable certificate (proof).

Equivalently: Problems solvable by a **nondeterministic Turing machine** in polynomial time.

Key: Every problem in **P** is also in **NP**, because if you can solve it, you can certainly verify a solution quickly.

Open question: Is **P = NP**?

Example problems:

- Boolean satisfiability (SAT): Given a Boolean formula, is there an assignment of variables that makes it true?
- Hamiltonian path: Is there a path visiting each vertex exactly once?
- Graph coloring: Can the vertices be colored with (k) colors so no adjacent vertices share a color?

NP-Hard:

A problem is **NP-hard** if **every problem in NP** can be **reduced** to it in **polynomial time**.

- This means it is at least as hard as the hardest problems in NP.
- **NP-hard problems need not be in NP** themselves (can be harder, e.g., not decision problems, or requiring exponential verification).
- An NP-hard decision problem that is also in NP is called **NP-complete**.

Example NP-hard problems not necessarily in NP:

- Halting problem (undecidable, trivially NP-hard by some definitions, but classic reductions use decision problems in NP).
- Optimization versions of NP-complete problems: e.g., "Find the shortest traveling salesman tour" (not a yes/no question).

NP-Complete:

A problem is **NP-complete** if:

1. It is in **NP**.
2. It is **NP-hard**.

Thus, NP-complete problems are the **hardest problems in NP**.

If **any** NP-complete problem can be solved in polynomial time, then **P = NP**.

First NP-complete problem (Cook–Levin theorem, 1971):

Boolean satisfiability (SAT) is NP-complete.
Other famous NP-complete problems:

- 3-SAT
- Traveling salesman (decision version: "Is there a tour of length ≤ k?")
- Graph coloring
- Clique problem
- Vertex cover

Relationships (Visual Summary):

- **P ⊆ NP** (likely **P ≠ NP**, unproven).
- **NP-complete = NP ∩ NP-hard**.
- **NP-hard** includes **NP-complete** and possibly harder problems (even outside NP).

```
All problems
│
├── NP-Hard (includes NP-Complete and harder)
│   │
│   └── NP-Complete = NP ∩ NP-Hard
│        │
├── NP (problems with polynomial-time verification)
│   │
└── P (problems with polynomial-time solving) ⊆ NP
```

Key implication:
If you find a polynomial-time algorithm for **any** NP-complete problem, then **P = NP**.
No such algorithm has been found despite decades of effort, so most experts believe **P ≠ NP**.

02. Prove that 3SAT problem is NP-complete.

1. **3SAT is in NP**
2. **3SAT is NP-hard** (by reducing **SAT** to **3SAT** in polynomial time).

Step 1: 3SAT ∈ NP

A 3SAT instance is a Boolean formula in **conjunctive normal form (CNF)** where each clause has **exactly 3 literals**.
Example clause: $(x_1 \lor \neg x_2 \lor x_3)$.
Given:

- A 3CNF formula ϕ with n variables and m clauses.
- A certificate: an assignment of truth values to the variables.

Verification:
Check each clause of ϕ under the given assignment.

- Each clause has 3 literals ⇒ checking one clause takes $O(1)$ time.
- Checking all (m) clauses takes $O(m)$ time, polynomial in input size.

If all clauses are satisfied ⇒ output YES; else NO.
Thus, **3SAT ∈ NP**.

Step 2: 3SAT is NP-hard

We prove by showing **SAT $\leq_p$ 3SAT** (SAT polynomial-time reduces to 3SAT).

SAT Problem:

Input: Boolean formula ϕ in CNF (clauses can have any number of literals).
Question: Is ϕ satisfiable?

Reduction idea:

Transform each clause C of ϕ into a set of **3-literal clauses** such that satisfiability is preserved.

Case analysis for clause C:Let C = (l_1 ∨ l_2 ∨ ....l_k) with k literals.

Case 1: k = 1

Example: C = (x).

Introduce **two new variables** a, b not used elsewhere.

Replace C with: [(x ∨ a ∨ b) ∧ (x ∨ a ∨ ¬ b) ∧ (x ∨ ¬ a ∨ b) ∧ (x ∨ ¬ a ∨ ¬ b)]

Check:

The only way to satisfy all 4 clauses is to set x = True (since setting x = False) would require choosing a, b to satisfy all, impossible — try it).

Case 2: k = 2

Example: C = (x ∨ y).

Introduce **one new variable** z.

Replace C with: [(x ∨ y ∨ z) ∧ (x ∨ y ∨ ¬ z)]

Check:

To satisfy both clauses, (x ∨ y) must be True (because if x ∨ y is False, both clauses are False regardless of z).

Case 3: k = 3

Example: C = (x ∨ y ∨ z).

Already a 3-literal clause ⇒ keep unchanged.

Case 4: (k ≥4)

Example: C = (ℓ1 ∨ ℓ2∨......... ∨ ℓk).

Introduce k-3 **new variables** z1, z2,......., zk-3.

(ℓ1 ∨ℓ2 ∨ z1) ∧(¬z1 ∨ℓ3 ∨z2)∧(¬z2 ∨ℓ4 ∨z3)∧⋯∧(¬zk−4 ∨ℓk−2 ∨zk−3)∧(¬zk−3 ∨ℓk−1 ∨ℓk)

This is called the **standard reduction**.

Why it works:

- Suppose C is satisfied in original formula ⇒ some ℓj=True.

Set z1 ,...,zt−1 =True for clauses before ℓj, and zt ,⋯=False for clauses

Then all new clauses are satisfied.

- Conversely, suppose all new clauses are satisfied.

If z1 is False ⇒ first clause forces ℓ1 ∨ℓ2 =True.

If z1 is True, look at second clause: if z2 is False ⇒ ℓ3 =True, else continue.

Eventually, if all z's are True, last clause forces ℓk−1 ∨ℓk =True.

So at least one original literal in C is True.

Polynomial time:

Each clause of length k becomes at most k-2 new 3-literal clauses.

Total new clauses ≤ ∑ (k_i - 2) ≤ O(length of original formula).

New variables are per clause, easily managed.
Transformation done in **linear time** per clause ⇒ polynomial overall.

Conclusion

1. **3SAT ∈ NP** (easy verification).
2. **SAT $\leq_p$ 3SAT** (by above reduction).
3. **SAT is NP-complete** (Cook–Levin theorem).
4. Therefore, **3SAT is NP-hard** (because any NP problem reduces to SAT, SAT reduces to 3SAT, so any NP problem reduces to 3SAT by composition).
5. Since 3SAT is in NP and is NP-hard ⇒ **3SAT is NP-complete**.

SAT is NP-complete

03. Given two problems Π1,Π2 with Π1≤pΠ2Π1, and Π2 solvable in O(n^k) with reduction in O(n), show Π1 solvable in O(n^k). (Polynomial-time reduction complexity.)

1. Understanding the problem setup

We have: Π1 ≤p Π2

- This means: there is a polynomial-time reduction from Π1 to Π2 , i.e., There exists a polynomial-time computable function (f) such that:

  $$x \in \Pi 1 \iff f(x) \in \Pi 2$$

  for all input x.
- The reduction runs in **O(n)** time (where n = |x|).
- Π2 is solvable in **O(m^K)** time (where m is the input size for Π2).

We want to show Π2 is solvable in **O(n^k)** time.

2. Step-by-step solution

Let's formalize:

1. Let x be an input to Π1, with size n = |x|.
2. Since Π1 ≤p Π2 in O(n) time, we can compute f(x) in O(n) time.

Let m = |f(x)| = size of the instance of Π2.
Since the reduction runs in O(n) time, the output f(x) can be at most **O(n) size** (because in O(n) steps, you can write at most O(n) symbols, assuming a reasonable encoding model).
More formally: in polynomial-time reductions, the output size is bounded by a polynomial in (n). Here it's linear time, so m <= cn for some constant c.

3. Therefore m = O(n).
4. Now we solve Π2on input f(x). This takes O(m^k) time.
5. Since m = O(n), O(m^k) = O((cn)^k) = O(n^k).

3. Total time for Π1

- Reduction time: O(n)
- Solving Π2 time: O(n^k)

Total: O(n) + O(n^k) = O(n^k) (since k >=1 for polynomial time).
Therefore Π1 is solvable in O(n^k) time.

04. Decide truth of: if Π∈NP and Π≤pΠ'where Π' is NP-complete, then Π is NP-complete.

The statement is NOT always true.

It is not sufficient for a problem Π to be in NP and reducible to an NP-complete problem to conclude that Π is NP-complete.

Why?

To prove a problem Π is NP-complete, we need:

$$\Pi \in NP$$

Every NP problem reduces to Π (i.e., an NP-hardness proof)

But the given condition only provides:

Π∈NP and Π≤p Π', where Π' is NP-complete.

Counterexample

Take an easy problem, like:

Π = "Is the input string length even?"

This is solvable in linear time → so Π∈P⊆NP.

We can reduce it to SAT (NP-complete) trivially:

Output a fixed satisfiable formula for even-length inputs and an unsatisfiable one for odd inputs.

$$\text{So: } \Pi \le_p \text{SAT}$$

BUT Π is not NP-hard and definitely not NP-complete.

Correct Direction

The reverse direction is the one used to prove NP-completeness:

$$\Pi' \le_p \Pi \text{(where } \Pi' \text{ is NP-complete)}$$

This means Π is at least as hard as an NP-complete problem.

✅ Final Verdict

False.

From Π∈NP and Π≤p Π' where Π' is NP-complete, we cannot conclude Π is NP-complete.

We must instead show Π'≤p Π.

04. Show that if L1≤pL2 and L2≤pL3, then L1≤pL3.

Proof of transitivity of polynomial-time reductions

We want to show:

L1 ≤p L2 and L2≤pL3 , then L1≤pL3 .

Step 1: Understanding the definition of $\le_p$

A≤p B means:

There exists a **polynomial-time computable** function f such that:

$$x \in A \Longleftrightarrow f(x) \in B$$

for all inputs x.

Step 2: What is given

1. $L1 \le_p L2 \Rightarrow \exists$ function f computable in polynomial time,
2. such that $x \in L1 \Leftrightarrow f(x) \in L2$.
3. $L2 \le_p L3 \Rightarrow \exists$ function g computable in polynomial time,

such that $y \in L2 \Leftrightarrow g(y) \in L3$.
We need to construct a polynomial-time computable function h such that
$x \in L1 \Leftrightarrow h(x) \in L3$.

Step 3: Construct the reduction from L1 to L3

Define $h(x) = g(f(x))$.

- **Correctness**:

$$x \in L1 \Leftrightarrow f(x) \in L2 \text{ (by reduction } L1 \le_p L2 \text{)}$$
$$\Leftrightarrow g(f(x)) \in L3 \text{ (by reduction } L2 \le_p L3 \text{)}$$

Thus $x \in L1 \Leftrightarrow h(x) \in L3$.

Step 4: Show h is polynomial-time computable

We have:

1. Computation of $f(x)$ takes time $p1(|x|)$ for some polynomial $p1$.

Let $m = |f(x)|$. Because f runs in polynomial time, the output size m is bounded by a polynomial in $|x|$, say $m \le q(|x|)$ for some polynomial q.

2. Computation of g on input of size m takes time $p2(m)$ for some polynomial $p2$.

Therefore, total time for $h(x) = g(f(x))$:

- Step 1: compute $f(x)$ in time $p1(|x|)$, producing output of size $m \le q(|x|)$.
- Step 2: compute g on input of size m in time $p2(m) \le p2(q(|x|))$.

Total time $\le p1(|x|) + p2(q(|x|))$, which is polynomial in $|x|$, because the sum/composition of polynomials is a polynomial.

Step 5: Conclusion

We have constructed a polynomial-time computable function h satisfying
$x \in L1 \Leftrightarrow h(x) \in L3$, so L1≤pL3L1 $\le_p$ L3 .

Transitivity holds.

This transitivity property is crucial for building the "reduction chain" in NP-completeness proofs, allowing us to conclude that if $A \le_p B$ and B is NP-hard, then A is NP-hard.

05. Two problems P1,P2 are polynomial-time equivalent if P1≤PP2 and P2≤PP1; prove/disprove that every two NP-Complete problems are polynomial-time equivalent.

Claim: Every two NP-complete problems are polynomial-time equivalent.

Step 1: Understanding the statement

Polynomial-time equivalence means:
For two problems P1, P2 , we say $P1 \equiv_p P2$ if:

$$P1 \le_p P2 \text{ and } P2 \le_p P1$$

is polynomial-time reduction.

Step 2: What does NP-complete mean?

- A problem P is **NP-complete** if:
  - $P \in NP$
  - P is **NP-hard**: For every problem $Q \in NP$, $Q \le_p P Q \le_p P$.

Step 3: Take two arbitrary NP-complete problems P1 and P2

We must check if $P1 \le_p P2$ and $P2 \le_p P1$ both hold.
Since P1 is NP-complete, it is NP-hard ⇒ every problem in NP reduces to P1.
In particular, $P2 \in NP$ (because NP-complete $\subseteq$ NP), so:

$$P2 \le_p P1$$

Since P2 is NP-complete, it is NP-hard ⇒ every problem in NP reduces to P2 .
Since $P1 \in NP$:

$$P1 \le_p P2$$

Thus:

$$P1 \le_p P2 \text{ and} P2 \le_p P1$$

Therefore $P1 \equiv_p P2$ by definition.

Step 4: Conclusion

The statement is **true**. All NP-complete problems are polynomial-time equivalent.

True

06.Define 3SAT and Vertex Cover decision problem.

3SAT (3-Satisfiability) Decision Problem

Instance:

A Boolean formula F in **3-Conjunctive Normal Form (3-CNF)**, meaning:

$$F = C1 \wedge C2 \wedge \cdots \wedge Cm$$

where each clause Ci contains **exactly three literals**, and each literal is either a variable xj or its negation $\bar{x}j$

Question:

Does there exist a truth assignment to the variables such that the whole formula evaluates to **TRUE**?

Vertex Cover (VC) Decision Problem

Instance:

An undirected graph G = (V, E) and a positive integer k.

Question:

Does there exist a subset $C \subseteq V$ with $|C| \le k$ such that **every edge in E** has **at least one endpoint** in C?

That is, C "covers" all edges.

✓ Both are decision problems.

✓ 3SAT is NP-complete.

✓ Vertex Cover is NP-complete.