Compare P and NP class problems, Justify that P is a subset of N.

These classes relate to the *time complexity* of solving computational problems, specifically *decision problems* (problems with a yes/no answer), relative to the size of the input.[1]

Class P (Polynomial Time)

1. **Definition:** P is the set of decision problems that can be **solved** by a **deterministic Turing machine** in **polynomial time**.[2]
2. **Explanation:**
    - **Deterministic Turing Machine:** This is a standard model of computation that follows a single, predetermined path of execution for a given input. Think of a regular computer program without randomness.
    - **Polynomial Time:** The time (number of computational steps) required by the algorithm to find the solution is bounded by a polynomial function of the input size 'n'. This means the running time is O(nk) for some constant k.[3]
    - **"Solved":** The algorithm must find the correct yes/no answer.
3. **Intuition:** Problems in P are generally considered "efficiently solvable" or "tractable."[4] As the input size grows, the time required to solve the problem grows reasonably (polynomially), not exponentially.
4. **Examples:**
    - Determining if a number is prime (AKS primality test).
    - Finding the shortest path between two nodes in a graph (Dijkstra's algorithm).
    - Sorting a list.
    - Checking if a graph is connected.

Class NP (Nondeterministic Polynomial Time)

1. **Definition:** NP is the set of decision problems for which a proposed solution (a "certificate" or "witness") can be **verified** by a **deterministic Turing machine** in **polynomial time**.
2. **Explanation:**
    - **"Verified":** If the answer to the decision problem is "yes," there must exist some certificate (extra piece of information) that allows you to quickly (in

polynomial time) check that the answer is indeed "yes." If the answer is "no," no certificate should make the verifier output "yes."
    - **Certificate:** This is a potential solution or proof. For example, for the Traveling Salesperson Problem (decision version: "Is there a tour with total length less than L?"), the certificate would be the specific sequence of cities forming the tour.
    - **Polynomial Time Verification:** The algorithm to check the certificate must run in polynomial time (O(nk)) relative to the size of the *original problem input* 'n'.[5]
3. **Alternative Definition:** NP is also the set of decision problems that can be solved by a *nondeterministic* Turing machine in polynomial time. A nondeterministic machine can explore multiple computational paths simultaneously (like guessing the right path). This definition is equivalent to the verification-based one.
4. **Intuition:** Problems in NP are those where solutions might be hard to *find*, but they are easy to *check* if someone gives you a potential solution.
5. **Examples:**
    - **Satisfiability (SAT):** Given a Boolean formula, is there an assignment of TRUE/FALSE values to variables that makes the whole formula TRUE? (Certificate: The satisfying assignment).
    - **Traveling Salesperson Problem (Decision Version):** Given a list of cities and distances, is there a tour visiting all cities exactly once with a total length less than L? (Certificate: The tour sequence).
    - **Vertex Cover (Decision Version):** Given a graph and an integer k, is there a set of k or fewer vertices such that every edge in the graph is incident to at least one vertex in the set? (Certificate: The set of vertices).
    - **Integer Factorization (Decision Version):** Given integers n and m, does n have a factor d such that 1<d<m? (Certificate: The factor d).

Comparison: P vs NP

Feature	**Class P**	**Class NP**
Core Idea	Problems efficiently **solvable**.	Problems efficiently **verifiable** (given a solution/certificate).

Time Complexity	Solution found in polynomial time (O(nk)).	Solution verified in polynomial time (O(nk)).
Machine Model	Deterministic Turing Machine (solving).	Deterministic Turing Machine (verifying) / Nondeterministic TM (solving).
Difficulty	Considered "easy" or "tractable".	May be "hard" to solve, but "easy" to check.
Key Question		Does P=NP? (Can every problem verified quickly also be solved quickly?)

Justification that P is a Subset of NP (P⊆NP)

To show that P is a subset of NP, we need to demonstrate that any problem solvable in polynomial time (i.e., any problem in P) is also verifiable in polynomial time (i.e., it meets the definition of being in NP).

Here's the argument:

1. **Take any problem X in P.** By definition, this means there exists a deterministic algorithm A that solves X in polynomial time, say O(nk) for input size n. Given an input w, algorithm A(w) outputs "yes" or "no" correctly in O(nk) time.

2. **We need to show that X is also in NP.** This requires us to construct a polynomial-time *verifier* algorithm V. The verifier V takes the problem input w and a potential certificate c. It must output "yes" if and only if w is a "yes" instance of the problem X (and some valid certificate exists).

3. **Construct the Verifier V:**

    - Let V be an algorithm that takes input w and certificate c.
    - Inside V, **completely ignore the certificate c**.
    - Run the original polynomial-time solving algorithm A on the input w.
    - If A(w) outputs "yes", then V(w,c) outputs "yes".
    - If A(w) outputs "no", then V(w,c) outputs "no".

4. **Verify the Verifier's Properties:**
    - **Correctness:** If w is a "yes" instance of problem X, algorithm A(w) outputs "yes", so V(w,c) will output "yes" (for any certificate c, including an empty one). If w is a "no" instance, A(w) outputs "no", so V(w,c) outputs "no" for all certificates c. This matches the requirement for an NP verifier.
    - **Efficiency:** The verifier V essentially just runs algorithm A. Since A runs in polynomial time (O(nk)), the verifier V also runs in polynomial time.
5. **Conclusion:** Since we can construct a polynomial-time verifier V for any problem X in P (by simply using the solver algorithm A and ignoring the certificate), it follows that every problem in P satisfies the definition of being in NP. Therefore, **P is a subset of NP (P⊆NP)**.[6]

The huge open question in computer science is whether P=NP or P=NP. Most researchers suspect P=NP, implying that there are problems whose solutions are easy to verify but genuinely hard to find. Proving this, however, remains one of the most significant unsolved problems in mathematics and computer science.

How NP Complete and NP Hard Problems are related with each other and How NP Hard problems can be converted to NP Complete problems.

NP-Hard Problems (2 Marks)

1. **Definition:** A problem H is **NP-hard** if every problem L in the complexity class NP can be reduced to H in polynomial time (L≤pH).
2. **Meaning:** This signifies that H is "at least as hard as" any problem in NP. If you could find an efficient (polynomial-time) algorithm to solve *any* single NP-hard problem H, you could use the polynomial-time reduction to solve *every* problem in NP efficiently, implying P=NP.
3. **Key Point:** An NP-hard problem does **not** necessarily have to be in NP itself. It could be a problem for which solutions are not easily verifiable (like optimization problems asking for the *best* solution) or even undecidable problems (like the Halting Problem).

NP-Complete Problems (2 Marks)

1. **Definition:** A problem C is **NP-Complete** (NPC) if it meets two conditions:
    - **(a) C is in NP:** There exists a polynomial-time algorithm to *verify* if a proposed solution (certificate) for a "yes" instance is correct.
    - **(b) C is NP-hard:** Every problem L in NP is polynomial-time reducible to C (L≤pC).
2. **Meaning:** NP-Complete problems are the **hardest problems within the class NP**. They are both verifiable quickly (in NP) and are such that any other NP problem can be transformed into them quickly (NP-hard).
3. **Significance:** They sit at the frontier of the P vs NP question. Finding a polynomial-time algorithm for *any* NPC problem would prove P=NP. Most researchers believe P ≠ NP, implying that no NPC problem has a polynomial-time solution. Examples include SAT, 3-SAT, Traveling Salesperson (decision version), and Vertex Cover (decision version).

How NP Complete and NP Hard Problems are related with each other and How NP Hard problems can be converted to NP Complete problems.

1. Relationship between NP-Complete and NP-Hard Problems

- **NP-Hard:** A problem H is NP-hard if *every* problem L in NP can be reduced to H in polynomial time (L≤pH). This essentially means H is "at least as hard as" any problem in NP. If you find a polynomial-time algorithm for H, you can solve all problems in NP in polynomial time (implying P=NP). Importantly, an NP-hard problem **does not** have to be in NP itself.

- **NP-Complete (NPC):** A problem C is NP-Complete if it satisfies *two* conditions:

    - C is in NP (meaning a solution can be *verified* in polynomial time).[1]
    - C is NP-hard (meaning every problem in NP reduces to C in polynomial time).
- **The Relationship:**

    - **NP-Complete is a subset of NP-Hard:** By definition, every NP-Complete problem *must* be NP-hard. The NP-hard property is one of the two requirements for being NP-Complete.
    - **NP-Hard is NOT necessarily NP-Complete:** There are NP-hard problems that are not NP-Complete.[2] This occurs when an NP-hard

problem does *not* satisfy the first condition of NPC – that is, the problem is not itself in NP.

- **In summary:** Think of NP-Hard as a broad category of problems that are at least as hard as anything in NP. NP-Complete is a more specific category, containing only those NP-hard problems that are *also* members of NP.[3]

    - *Example NP-Complete (in NP & NP-hard):* SAT, 3-SAT, Vertex Cover (decision version), Traveling Salesperson Problem (decision version: Is there a tour of length ≤K?).[4]
    - *Example NP-Hard but not NP-Complete:*
        - *TSP Optimization:* Find the length of the *shortest* tour. This isn't a decision (yes/no) problem, so it's not in NP, but it is NP-hard.
        - *The Halting Problem:* Determining if a program will ever stop.[5] This problem is undecidable, meaning no algorithm can solve it for all inputs, so it's certainly not in NP, but it is NP-hard.

2. How NP-Hard Problems Can Be "Converted" to NP-Complete Problems

You don't typically "convert" an NP-hard problem into an NP-Complete one in the sense of changing the problem itself. Instead, the question usually relates to: **What does it take for an NP-hard problem to *also* be NP-Complete?**

Based on the definitions above, if you have a problem H that you already know is NP-hard, to show it is NP-Complete, you only need to satisfy the *other* condition:

- **Prove that H is in NP.**

To prove H is in NP, you must show:

1. H is a **decision problem** (it has a "yes" or "no" answer).
2. For any instance of H where the answer is "yes", there exists a **certificate** (a proof or witness).
3. There is a **polynomial-time algorithm** (a verifier) that can take the problem instance and the certificate, and correctly confirm ("verify") that the answer is indeed "yes".

Common Scenarios:

- **NP-Hard Decision Problems:** If an NP-hard problem H is already a decision problem, the "conversion" to proving it's NP-Complete simply involves constructing and analyzing the polynomial-time verifier to show $H \in NP$. Many problems are proven NP-hard first (e.g., via reduction from SAT), and then separately shown to be in NP.
- **NP-Hard Optimization Problems:** Often, an NP-hard problem is an *optimization* problem (e.g., find the minimum cost, maximum value).[6] Optimization problems are not decision problems and thus not in NP. However, they often have a closely related *decision version* which *can* be NP-Complete.
    - *Example:* TSP Optimization (NP-hard: find shortest tour length) vs. TSP Decision (NPC: is there a tour $\leq K$?). The decision version is in NP (certificate: the tour, verifier: check length $\leq K$). Proving the decision version is also NP-hard makes it NP-Complete. So, we didn't "convert" the optimization problem; we formulated a related decision problem that *is* NP-Complete.
- **NP-Hard Problems Outside NP:** Problems like the Halting Problem are NP-hard but are provably not in NP (they are undecidable).[7] These can *never* become NP-Complete.

Conclusion: An NP-hard problem becomes NP-Complete if and only if it is also in NP. For decision problems, this requires proving polynomial-time verifiability. For optimization problems, this often involves considering the corresponding decision problem, which might then be proven NP-Complete.

Notes on Computational Complexity Theory

1. Introduction to Complexity Theory

- Complexity theory is about determining **what computers can and cannot do**.
- It involves classifying problems based on the time and resources required to solve them.
- Humans have always wanted to do things fast and always want to improve algorithms.

2. Problem Classification Overview

Sources classify problems and algorithms in several ways:

- **Computable vs. Non-Computable Problems**:
    - A problem is **Computable** if there is an algorithm that solves it in a finite number of steps (finite time).
    - A problem is **Non-Computable** if no such algorithm exists, meaning any process would potentially run in infinite steps.
    - Example of a Non-Computable problem: The **Halting Problem** (given an algorithm and its input, determine if it will halt or run forever).
- **Tractable vs. Intractable Problems**:
    - Problems solvable by a **Deterministic Turing Machine (DTM)** in **Polynomial time** are considered **Tractable**.
    - Problems solvable by a **DTM** in **Exponential time** are considered **Non-Tractable**.
- **Problem Types**:
    - **Decision Problems**: Problems with a yes/no answer.
    - **Optimization Problems**: Problems seeking the "best" solution (e.g., shortest path, finding the best score). Optimization problems often have a "verifiable path". Finding the best solution might be difficult, but verifying a proposed best solution is often easier.
    - Optimization problems can sometimes be converted into Decision problems. For example, the TSP Optimization problem (find the shortest tour) can be converted to a TSP Decision problem ("Is there a tour with length less than k?").
    - **Search Problems**: Finding a specific element or solution.

3. Time Complexity Classes

- **Polynomial Time (P)**: Algorithms or problems solvable in time proportional to a polynomial function of the input size (e.g., O(n), O(n^2), O(n^k) for some constant k).
    - Often associated with Tractable problems.
    - Examples mentioned as potentially solvable in polynomial time in the sources include:
        - Line Search
        - Insertion Sort

        - SAT (Satisfiability) - *Note: Source lists SAT as taking exponential time, and sources classify it as NP-Hard/NP-Complete. Source also lists TSP, Sudoku, and Candy Crush as solvable in polynomial time.*
        - TSP (Traveling Salesperson Problem) - *Note: Sources and classify TSP differently (Non-Tractable/NP-Hard).*
        - Sudoku
        - Candy Crush
- **Exponential Time**: Algorithms or problems requiring time proportional to an exponential function of the input size (e.g., O(2^n)).
    - Often associated with Non-Tractable problems.
    - Examples mentioned include:
        - Towers of Hanoi?
        - SAT - *Note: Contradicts and.*
        - Sum of Subsets
        - Problems involving searching a large state space.

4. Deterministic vs. Nondeterministic Algorithms

- **Deterministic Algorithms (DTM)**:
    - The path of computation is always clear and known.
    - Follow a single computational path.
    - If choice is involved, the choice is known or follows a determined rule.
    - Deterministic algorithms generally take polynomial time *for P problems - Note: Sources and indicate DTMs can take exponential time for Non-Tractable problems.*
- **Nondeterministic Algorithms (NDTM)**:
    - Do not follow a single path; can follow multiple paths simultaneously.
    - Involve "guessing" or randomly choosing from possibilities. The "choice" function in an NP algorithm selects elements without a known method. Choice can be 0 or 1.
    - Described as knowing "some" path, but potentially unable to solve all instances.
    - Can "guess the correct solution" in Polynomial time.
    - Nondeterministic algorithms are considered faster than deterministic ones.
    - An NP algorithm typically has three functions: **Choice**, **Success** (if a solution is found), and **Failure** (if no solution is found).

    - The complexity of a nondeterministic algorithm for searching an element is O(n).
    - If the "choice" in a nondeterministic algorithm becomes a known member, the algorithm becomes deterministic.

5. Problem Classes (P, NP, NP-Hard, NP-Complete)

- **P (Polynomial Time)**: The class of decision problems solvable by a **Deterministic Turing Machine (DTM)** in **Polynomial time**. These are generally considered Tractable.
- **NP (Nondeterministic Polynomial Time)**:
    - The class of decision problems solvable by a **Nondeterministic Turing Machine (NDTM)** in **Polynomial time**.
    - Equivalently, problems where a proposed solution can be **verified** in **Polynomial time** by a **DTM**.
    - Many famous NP problems exist.
- **NP-Hard**:
    - A problem 'H' is NP-Hard if **every problem in NP can be reduced to H in polynomial time**.
    - This means if you could solve H in polynomial time, you could solve *all* problems in NP in polynomial time via reduction.
    - If it's impossible to determine a polynomial time algorithm for a problem, it's likely NP-Hard.
    - Examples of problems listed as NP-Hard: **SAT, Knapsack, TSP, Sum of Subsets, Graph Coloring, Hamiltonian Cycle**.
- **NP-Complete**:
    - A problem 'C' is NP-Complete if it satisfies two conditions:
        1. It is **NP-Hard**.
        2. It is in the class **NP** (i.e., solvable by NDTM in polynomial time, or verifiable by DTM in polynomial time).
    - Essentially, these are the "hardest" problems in NP. If any NP-Complete problem can be solved in polynomial time, then P=NP.
    - Example: **SAT (Satisfiability)** is NP-Complete. If SAT can be solved in P time, then any NP-Hard problem can also be solved in P time via reduction.
    - A problem is NP-Complete if it reduces to SAT *and* has an NP algorithm.

    - The provided sources mention an NP algorithm exists for SAT, called Colledia.

Relationships between Classes:

- NP-Complete problems are a subset of NP-Hard problems.
- NP-Complete problems are also in NP.
- The relationship between P and NP is unknown (P=NP? or P != NP?).

6. Key Concepts Explained

- **Reduction**:
    - A method to solve NP problems.
    - Relating one problem to another, often based on their structure or size.
    - If problem A can be "reduced" to problem B in polynomial time, it means an algorithm for B can be used to solve A (perhaps with some polynomial overhead).
    - This concept is used to classify problems and prove their hardness. If problem A reduces to problem B, and B is known to be hard (like NP-Hard), then A is also hard.
    - Example: Magic Square can be reduced to Tic-Tac-Toe. Solving Tic-Tac-Toe can then solve Magic Square and also Number Scrabble.
    - Example: 3CNF (a specific logical formula structure) can be reduced to SAT.
    - Example: SAT can be reduced to Knapsack. If SAT is in P, Knapsack is in P.
- **P vs NP Question**:
    - The question of whether every problem whose solution can be quickly verified (NP) can also be quickly solved (P).
    - Whether P is equal to NP (P=NP) or P is a proper subset of NP (P != NP).
    - There is no proof yet for either case.
    - If P=NP were proven true, it would mean all NP-Complete and NP-Hard problems could be solved in polynomial time, making them tractable. This would imply many currently considered hard problems are actually easy to solve quickly.

7. Specific Problems and their Classifications (as per sources)

- **Polynomial Time / Tractable (P)**:
    - Line Search
    - Insertion Sort
    - SAT - *Contradicted by*
    - TSP - *Contradicted by*
    - Sudoku
    - Candy Crush
    - Tic-Tac-Toe (and related Magic Square, Number Scrabble via reduction)
- **Exponential Time / Non-Tractable**:
    - Towers of Hanoi?
    - SAT - *Contradicted by*
    - Sum of Subsets - *Contradicted by*
- **Non-Computable**:
    - Halting Problem
- **NP-Hard**:
    - SAT
    - Knapsack
    - TSP
    - Sum of Subsets
    - Graph Coloring
    - Hamiltonian Cycle
- **NP-Complete**:
    - **SAT (Satisfiability) - Proven by Cook's Theorem**. Satisfiability is a decision problem.

Contradictions in Sources: It's important to note the apparent contradictions in how some problems are classified across the sources. For example, SAT and TSP are listed as solvable in polynomial time and potentially solvable in P time via reduction, but also listed as taking exponential time and classified as NP-Hard/NP-Complete. Sum of Subsets is listed as exponential time and NP-Hard. These conflicting classifications within the provided text should be acknowledged.

8. Cook's Theorem

- **Cook's Theorem states that SAT is NP-Complete**.

- This is a foundational result because it means that if SAT can be solved by a deterministic algorithm in polynomial time (i.e., if SAT is in P), then every problem in NP can be solved by a deterministic algorithm in polynomial time (i.e., P=NP).

These notes cover the key concepts, problem types, complexity classes, and specific examples discussed in the provided sources, highlighting their relationships and the central questions in complexity theory like P vs NP. Remember to also review the specific examples and how reduction applies to them, as this is a key technique discussed.