Complexity Classes and Complete Problems

Rishika Aggarwal

Euler Circle

July 11, 2025

Overview

What is Complexity Theory?

Complexity Theory is the branch of theoretical computer science that studies:

1. How difficult computational problems are.
2. How resources like time and memory affect problem-solving.
3. Why some problems are easy to check but hard to solve.

Why does complexity theory matter?

1. Complexity theory studies what problems can be solved efficiently.
2. It organizes problems into classes like P, NP, $PSPACE$, etc.

Example: Checking a Sudoku solution can be done in seconds, but solving it may take a large amount of time. This is the kind of questions complexity theory answers.

Turing Machine (Definition)

A Turing Machine is a 7-tuple, $(Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$, where Q, Σ, Γ are all finite sets.

1. Q is the finite set of states the machine can be in.
2. Σ is the input alphabet not containing the blank symbol $\sqcup$.
3. Γ is the tape alphabet, where $\sqcup \in \Gamma$ and $\Sigma \subseteq \Gamma$.
4. δ is the transition function.
5. $q_0 \in Q$ is the start state of the machine.
6. $q_{\text{accept}} \in Q$ is the accept state.
7. $q_{\text{reject}} \in Q$ is the reject state, where $q_{\text{reject}} \neq q_{\text{accept}}$.

The Church-Turing Thesis states that any computation that can be performed by an algorithm can also be performed by a Turing machine.

Informally, a Turing machine is a simplified model of how a computer works. It has an infinite tape (like memory) and follows a set of rules to read and write symbols on that tape. It works one step at a time. It can describe any algorithm a real computer could execute. When we talk about how much time or space a problem takes to solve, we usually imagine how a Turing Machine would do it.

What is NP?

A problem is in NP if a solution can be verified in polynomial time.
Formal definition: A language $L \subseteq \{0,1\}^*$ is in NP if:

$$x \in L \iff \exists u \in \{0,1\}^{p(|x|)} \text{ such that } M(x,u) = 1$$

for some polynomial-time machine M.

Intuition:

- It's like checking a Sudoku or crossword puzzle.
- Someone hands you a filled grid (a "certificate").
- You check if it's correct in polynomial time.

Examples: SAT and $3-SAT$

NP-Completeness and Reductions

NP-Complete: For a problem A to be NP-complete, it should be in NP ($A \in NP$) and it should be NP-hard.

NP-Hard: A problem is NP-hard if it is as hard as the hardest problems in NP. This means that it is NP-hard if every problem in NP can be reduced to it in polynomial time.

Reduction: Converting one problem into another. If A is polynomial time reducible to B ($A \leq_p B$), solving B solves A.

What is *PSPACE*?

PSPACE: Problems that lie under *PSPACE* are solvable using a polynomial amount of memory.

$$PSPACE = \bigcup_{k \in \mathbb{N}} \text{SPACE}(n^k)$$

Even if it takes a long time, if memory usage stays polynomial, the problem is in *PSPACE*.

A problem is *PSPACE*-complete when

- The problem is in *PSPACE*
- The problem is *PSPACE*-hard.

Examples: *TQBF*, Formula Game

Latin Square Completion Problem

A Latin square of order n is an $n \times n$ grid filled with numbers from the set $\{1, 2, \ldots, n\}$, such that each number appears exactly once in each row and each column.

The Latin Square Completion (LSC) problem asks whether a partially filled Latin square can be completed into a full Latin square that still satisfies the row and column uniqueness conditions.

1	2	3	4
2	1	4	3
3	4	1	2
4	3	2	1

Figure: A Latin Square

Verification in NP:
Given a proposed completion of a Latin square, we can verify in polynomial time ($O(n)^2$) that:

- Each row contains all numbers from 1 to n exactly once,
- Each column contains all numbers from 1 to n exactly once,
- The completed entries agree with the original (non-blank) cells.

Therefore, Latin Square Completion $\in NP$.

Sudoku is NP-Complete

We consider the generalized Sudoku problem on an $n^2 \times n^2$ grid, where each cell must contain a digit from $\{1, 2, \ldots, n^2\}$.
Given a candidate filled grid (a certificate or a proposed solution), we can verify in polynomial time whether all constraints are satisfied. This verification can be done in the following times:

- Checking all n^2 rows takes $O(n^4)$ time.
- Checking all n^2 columns takes $O(n^4)$ time.
- Checking all n^2 blocks takes $O(n^4)$ time.

Thus, the verification of a solution can be performed in polynomial time with respect to the input size.
Therefore, Sudoku $\in NP$.

Now, using a Latin Square Completion problem, we construct a Sudoku instance as follows:

- Let the partially filled Latin square be denoted by a function:

$$L : \{1, \ldots, n\} \times \{1, \ldots, n\} \to \{1, \ldots, n\} \cup \{\square\}$$

Here, $L(i, j)$ is the value in row i, column j of the Latin Square.
If the cell is already filled, then $L(i, j) \in \{1, \ldots, n\}$
If the cell is blank, then $L(i, j) = \square$

- Then we construct a Sudoku grid S by embedding a Latin square inside it. Each entry $L(i, j)$ is placed into a specific sub-region of S, ensuring that row and column uniqueness from the Latin square are preserved. This helps us create a grid that also satisfies subgrids uniqueness.

- For each known entry $L(i, j) = a$, we fix the corresponding cell in the Sudoku grid S:

$$S[(i-1) \cdot n + 1, (j-1) \cdot n + 1] = a$$

This entry is treated as a fixed clue in the Sudoku puzzle.
For example, we embed a 3×3 Latin square into a 9×9 Sudoku grid.
If the Latin square has value 2 at position (2,3), then,
$(i-1) \cdot n + 1 = (2-1) \cdot 3 + 1 = 4$
$(j-1) \cdot n + 1 = (3-1) \cdot 3 + 1 = 7$
So, $S(4, 7) = 2$, which means that the number 2 will be in the cell (4,7) of the Sudoku grid.

- The remaining entries of the Sudoku are left blank. This still enforces the Latin Square completion properties.

Latin Square Embedded in Sudoku

1	2	3
3	1	**2**
2	3	1

Latin Square (3×3)

1			2			3		
3			1			**2**		
2			3			1		

Sudoku Grid (9×9)

Thus, any valid completion of the Sudoku corresponds to a valid Latin square completion. And conversely, a completed Latin square gives a valid Sudoku solution under this embedding.

This construction is computable in polynomial time, as we are only mapping positions and values between two grids of size $O(n^2)$.

Since Sudoku $\in NP$ and Sudoku is NP-hard, it is NP-complete.

Is Sudoku *PSPACE*-complete?

Recall that for a problem to *PSPACE*-complete, it should in *PSPACE* and it should be *PSPACE*-hard.
The following is a known complexity class hierarchy:

$$P \subseteq NP \subseteq PSPACE \subseteq PH.$$

Since Sudoku is in *NP*, and $NP \subseteq PSPACE$, it follows that Sudoku $\in PSPACE$.

Many *PSPACE*-complete problems involve:

- Two-player games with alternating moves,
- Long sequences of decisions with recursive or branching structure,
- Quantifier alternation, such as in *TQBF*.

Sudoku, by contrast, does not exhibit these traits:

- It is a single-player puzzle with no alternation or interaction.
- Once a candidate solution is proposed, it can be verified efficiently.
- It does not simulate universal or existential quantifier alternation.

This strongly suggests that Sudoku is not *PSPACE*-hard. However, we cannot formally prove that Sudoku is not *PSPACE*-complete unless we resolve the open question $NP \neq PSPACE$.

Open Problems in Complexity Theory

- **P vs NP:** Can we always find solutions as quickly as we can check them?
- **NP vs PSPACE:** Is solving problems with limited memory harder than just verifying answers?
- **PH Collapse:** Will adding more layers of complexity eventually stop giving us harder problems?

Thank you for your attention!