COMPLEXITY CLASSES AND COMPLETE PROBLEMS

RISHIKA AGGARWAL

Abstract. In this paper, we explore the foundational ideas of complexity theory by discussing major computational complexity classes, such as P, NP, and $PSPACE$. After reading the following paper, the reader will have a basic understanding of what it means for a problem to be hard for a computer to solve. To do this, we begin by introducing various complexity classes and the Turing machine. We then prove three major completeness theorems: SAT is NP-complete (Cook-Levin Theorem), $3-SAT$ is NP-complete, and $TQBF$ is $PSPACE$-complete. Along the way, we also explore how real-world problems like Sudoku fit into this theory, and discuss open questions such as if $P = NP$ and whether the Polynomial Hierarchy collapses. Finally, we touch on how these ideas connect to real-world topics like cryptography.

1. Introduction

Have you ever wondered why certain problems, like checking if a Sudoku solution is valid, feel easy, while actually solving them from scratch can be much harder, even for a computer? Questions like this are at the heart of computational complexity, a field of theoretical computer science that explores the limits of what computers can do, and how efficiently they can do it.

Theoretical computer science (TCS) emerged in the early 20th century through the groundbreaking work of mathematicians such as Alan Turing, Alonzo Church, and Kurt Gödel. Their work introduced formal models of computation, including the now famous Turing machine, which is a simple device that captures the essence of algorithmic problem-solving. This model still underlies how we reason about computation today.

As computers became more powerful, a new set of questions came up. These questions were broader, as they discussed how much time or memory it would take to solve a problem. This gave rise to complexity classes like P (problems solvable quickly), NP (problems where solutions are easy to check), and $PSPACE$ (problems solvable with limited memory). Understanding the relationships between these classes remains one of the deepest open challenges in theoretical computer science.

One key idea in complexity theory is the notion of completeness. A problem is called "complete" for a class if it is, informally, the hardest in that class. Solving a complete problem efficiently would mean we could solve every problem in the class efficiently. The

Date: 7th July, 2025.

Cook-Levin Theorem, which showed that the Boolean satisfiability problem (SAT) is NP-complete, was one of the first major results in this area and helped define how we think about computational difficulty today. Further historical context and contributions can be found in [FH03].

In this paper, we explore these fundamental ideas through definitions, theorems, and examples. We begin by formally defining complexity classes, reductions, and Turing machines. We then walk through classic results like the completeness of SAT, $3-SAT$, and $TQBF$. We also look at how puzzles like Sudoku fit into this framework, giving it a more real-world approach.

Finally, we highlight major open problems in the field, such as whether $P = NP$ or whether the Polynomial Hierarchy collapses. We also touch on the known separation between $PSPACE$ and $EXPTIME$, which is one of the few proven results about the structure of these classes.

2. Preliminaries

In this section, we define concepts that are fundamental to theoretical computer science and computational complexity. Additionally, we discuss time complexity and space complexity. This section forms the basis for understanding the related results discussed later in the paper. All the definitions are found in [Sip12, AB09]. This section aims to explain these definitions in simpler terms.

Definition 1: Turing Machine (TM). 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 set of states, which is the finite set of states the machine can be in.
(2) Σ is the input alphabet not containing the blank symbol $\sqcup$, which is the set of data the machine can read as input.
(3) Γ is the tape alphabet, where $\sqcup \in \Gamma$ and $\Sigma \subseteq \Gamma$, which includes all the symbols a Turing machine might read, write, or work with.
(4) $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$, is the transition function. This tells the machine what to do (which transition to make, what to write, and which direction to move).
(5) $q_0 \in Q$ is the start state, which is the state in which the machine begins.
(6) $q_{\text{accept}} \in Q$ is the accept state. This defines if the input is valid or accepted.
(7) $q_{\text{reject}} \in Q$ is the reject state, where $q_{\text{reject}} \neq q_{\text{accept}}$. This defines when a particular input fails the condition.

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

In simple words, 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.

Definition 2: Class P. Class P is the class of all languages that are decidable in polynomial time on a deterministic single tape Turing Machine.

$$P = \bigcup_k \text{TIME}(n^k)$$

In other words, it consists of all decision problems that can be solved by a deterministic Turing machine in polynomial time. The running time of such algorithms is bounded by some polynomial function of the input size.

Definition 3: The Class NP. A language $L \subseteq \{0,1\}^*$ is in NP if there exists a polynomial $p : \mathbb{N} \to \mathbb{N}$ and a polynomial-time Turing machine M such that for every $x \in \{0,1\}^*$,

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

If $x \in L$ and $u \in \{0,1\}^{p(|x|)}$ satisfy $M(x,u) = 1$, then we call u a certificate for x (with respect to the language L and the machine M).
Class NP includes decision problems for which a proposed solution can be verified in polynomial time. In other words, while the solution might be hard to find, it can be quickly checked once given. Many puzzles, such as Sudoku, fall into NP: solutions are difficult to compute from scratch, but easy to verify when someone gives you one.

Definition 4: Polynomial-Time Reduction. This is a general form of polynomial-time reduction, used to compare problem difficulty.
Let A and B be languages over the binary alphabet $\{0,1\}$. We say that A is polynomial-time reducible to B, written $A \leq_p B$, if there exists a polynomial-time computable function $f : \{0,1\}^* \to \{0,1\}^*$ such that for every string x:

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

That is, x is a yes-instance of A if and only if $f(x)$ is a yes-instance of B.

Definition 5: Reductions, NP-hardness, and NP-completeness. We say that a language $A \subseteq \{0,1\}^*$ is polynomial-time Karp reducible to a language $B \subseteq \{0,1\}^*$ denoted by $A \leq_p B$ if there is a polynomial-time computable function $f : \{0,1\}^* \to \{0,1\}^*$ such that for every $x \in \{0,1\}^*, x \in A$ if and only if $f(x) \in B$.
We say that B is NP-hard if $A \leq_p B$ for every $A \in NP$. We say that B is NP-complete if B is NP-hard and $B \in NP$.

A reduction is like converting one problem into another. If problem A can be reduced to problem B, it means that solving B helps us solve A too.

NP-hard problems are at least as hard as the hardest problems in NP.

NP-complete problems are in NP (verifiable quickly) and as hard as any other problem in NP.

Definition 6: Class $PSPACE$. $PSPACE$ is the class of languages that are decidable in polynomial space on a deterministic Turing Machine. In other words,

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

In simpler terms, $PSPACE$ is the set of problems that a computer can solve using a reasonable (polynomial) amount of memory, no matter how long it takes. Some problems in $PSPACE$ are hard to solve and may even be harder than those in NP.

Definition 7: $PSPACE$-Completeness. A language B is $PSPACE$-complete if it satisfies two conditions:
1. B is in $PSPACE$, and
2. Every other language, A, in $PSPACE$ is polynomial time reducible to B.
If only condition 2 is satisfied, we say that B is $PSPACE$-hard.
This is similar to NP-completeness.
Note that we use polynomial-time reductions for $PSPACE$-complete problems to keep things fair. The idea is to make sure that the reduction doesn't solve the hard part itself. It should just convert one problem into another without doing too much. This way, we know the second problem is at least as hard as the first one.

Definition 8: Polynomial Hierarchy. For every $i \geq 1$, a language L is in Σ_i^P if there exists a polynomial-time Turing Machine M and a polynomial q such that:

$$x \in L \iff \exists u_1 \in \{0,1\}^{q(|x|)} \forall u_2 \in \{0,1\}^{q(|x|)} \cdots Q_i u_i \in \{0,1\}^{q(|x|)} \; M(x, u_1, \ldots, u_i) = 1,$$

where Q_i is $\exists$ if i is odd, and $\forall$ if i is even.

We say that L is in Π_i^P if there exists a polynomial-time TM M and a polynomial q such that:

$$x \in L \iff \forall u_1 \in \{0,1\}^{q(|x|)} \exists u_2 \in \{0,1\}^{q(|x|)} \cdots Q_i u_i \in \{0,1\}^{q(|x|)} \; M(x, u_1, \ldots, u_i) = 1,$$

where Q_i denotes $\exists$ or $\forall$ depending on whether i is even or odd respectively.

The Polynomial Hierarchy is the set

$$PH = \cup_i \Sigma_i^P$$

In simpler terms, the Polynomial Hierarchy (PH) is like an extension of classes like P and NP. At each level, the machine solving the problem is allowed to alternate between "guessing" and "checking" a bounded number of times. These levels are defined using alternating quantifiers — like $\exists$ and $\forall$ — which means the complexity of problems increases as we go higher in the hierarchy.

Time Complexity. Time complexity describes how the running time of an algorithm grows with respect to the size of its input. For an input of size n, an algorithm is said to run in polynomial time if its running time can be bounded by a polynomial function of n, i.e., $O(n^k)$ for some constant k.
Below is a list of common time complexities and examples of problems or operations they are associated with:

Time Complexity	**Example or Typical Input Type**
$O(1)$	Constant-time operations, such as accessing a specific index in an array.
$O(\log n)$	Binary search in a sorted array.
$O(n)$	Linear scan of an array or list.
$O(n \log n)$	Efficient sorting algorithms like Merge Sort and Heap Sort.
$O(n^2)$	Nested loops on input of size n, such as matrix multiplication.
$O(n^3)$	Some dynamic programming and graph algorithms (e.g., Floyd-Warshall).
$O(2^n)$	Brute-force solutions to combinatorial problems, such as generating all subsets.
$O(n!)$	Algorithms that generate all permutations (e.g., solving the Traveling Salesman Problem by brute-force).

Space Complexity. Space complexity refers to the amount of memory an algorithm uses as a function of the size of the input n. It measures the maximum space (including the input, output, and working storage) that a Turing machine or algorithm needs to solve a problem. Below is a list of common space complexities and examples of problems or operations they are associated with:

Space Complexity	Description	Example Use Cases
$O(1)$	Constant space	In-place algorithms (e.g., array reversal)
$O(\log n)$	Logarithmic space	Binary search on an array
$O(n)$	Linear space	Storing an array or list of size n
$O(n \log n)$	Quasi-linear space	Merge sort (due to recursion stack)
$O(n^2)$	Quadratic space	Storing a graph adjacency matrix
$O(2^n)$	Exponential space	Memoization in some brute-force algorithms

3. NP-complete problems

This section discusses two fundamental proofs of NP-complete problems. These proofs are based on the proofs in [Tri24].

Theorem 1 (Cook-Levin Theorem): SAT is NP-complete.

Proof. To prove that SAT is NP-complete, we must show two things:
1. ($SAT \in NP$): SAT can be verified in polynomial time.
2. SAT is NP-hard: Every problem in NP can be reduced to SAT in polynomial time.

First, we prove $SAT \in NP$.
To show that SAT belongs to the class NP, we need to prove that for a given Boolean formula ϕ, if it is satisfiable, there exists a certificate (or proof) that can be verified in polynomial time.

A Boolean formula is said to be satisfiable if there exists at least one assignment of truth values (True or False) to its variables that makes the entire formula evaluate to True.

In the case of SAT, a certificate is simply a truth assignment to the variables in ϕ, the Boolean formula. This means assigning each variable a value of either True or False.

We can evaluate the formula by plugging in the values of True or False and checking whether the entire formula evaluates to True. This can be done in time proportional to the size of the formula, that is, in polynomial time with respect to the number of variables.

Since we can guess such an assignment non-deterministically, and verify its correctness in polynomial time, it follows that $SAT \in NP$.

Now we prove that SAT is NP-hard.
We must show that every problem in NP is polynomial-time reducible to SAT. That is, for any language $A \in NP$ (where A is the set of inputs that solve the problem), we want to reduce the question "Does input ω belong to A?" to the question "Is a certain Boolean formula ϕ satisfiable?"

Let $A \in NP$. By definition, this means there exists a non-deterministic Turing Machine (NTM) M that decides A in polynomial time. In simple words, every problem in the class NP has a non-deterministic Turing machine (NTM) that decides a language A, where A is the set of all inputs that solve the problem. This machine checks whether a given input ω belongs to A in polynomial time. Recall that problems in NP are those where solutions can be verified efficiently. The NTM does this by "guessing" a certificate and then verifying where it leads to acceptance. This guess is made by the machine itself. A non-deterministic machine is one that branches into all possible guesses and checks them parallelly.

Let $n = |\omega|$, where ω is the input string and n is the length of the input string. Since the NTM, which we will call M, runs in polynomial time, there must be a constant k for which the machine will halt at n^k. This constant depends on the problem.

Now, we will translate the behaviour of M into a Boolean formula, ϕ.

$$M \text{ accepts } w \iff \phi \text{ is satisfiable.}$$

ϕ should be satisfiable if and only if the machine M accepts the input ω.

To build ϕ, we need to represent the steps that M takes as it tries to accept ω. We need to record the whole process, that is every move the machine makes from the start to the end. This is done using a tableau.

A tableau is a table where each row represents the state of the machine at one specific time, including tape contents (what symbols are written on the tape), head position (the position M is reading/writing, and the current state (what it is currently doing). The first row is the initial configuration, and each subsequent row represents the next configuration. Basically, the table shows step-by-step how the machine runs on the input. Each row shows the full configuration of the machine at a given time step.

#	q_0	w_1	w_2	w_3	$\cdots$	w_n	$\sqcup$	$\sqcup$	$\cdots$	$\sqcup$	#
#	w_1'	q_1	w_2	w_3	$\cdots$	w_n	$\sqcup$	$\sqcup$	$\cdots$	$\sqcup$	#
#			$\vdots$			$\vdots$			$\vdots$		#
#					$\cdots$						#

This is what a sample tableau looks like. [Tri24]

Since M runs in at most n^k steps, and it can only use n^k cells on the tape of M, our tableau will be of size $n^k \times n^k$. Each cell of the tableau shows what is written on M's tape at a particular time and position. It shows one value and contains either a tape symbol, a state symbol, or a delimiter.

Here,

(1) Q: the set of states of M, like start, accept, reject, etc.

(2) Γ: the tape alphabet, like 0 or 1.

(3) $C = Q \cup \Gamma \cup \{\#\}$: the set of all possible symbols used in the tableau, including a delimiter.

We define a variable $x_{i,j,s}$ for each cell in the tableau. Here, i is the row number (time step), j is the column number (tape cell) and $s \in C$. The variable is true if and only if symbol s appears in row i, column j of the tableau ($[i, j]$).

Since the tableau is a square of size $n^k \times n^k$, we have n^{2k} total cells. So, our total number of variables is: $n^{2k} \times |C|$

Because $|C|$ (the number of possible symbols, including tape symbols, machine states, and delimiters) is fixed and does not grow with the size of the input, this total is still polynomial in n.

Our Boolean formula ϕ is made up of multiple parts. Each of these small parts ensures that the machine M is doing the step that it is supposed to do. All the small parts are combined by using AND ($\wedge$). So, for the formula to be true, everything has to be true. If this occurs, the tableau shows a complete and correct sequence of steps where the machine starts, follows the rules, and ends by accepting the input ω.Thus, ϕ being satisfiable is the same as the machine M accepting ω.

We define ϕ as the conjunction ($\wedge$) of four smaller formulae.
Specifically, we construct $\phi = \phi_{\text{start}} \wedge \phi_{\text{cell}} \wedge \phi_{\text{accept}} \wedge \phi_{\text{move}}$, where:

(1) Start Condition (ϕ_{start}): This condition ensures that the first row of the tableau correctly represents the starting configuration of M on the input ω. As in the sample tableau, the first row begins with a delimiter symbol (#), followed by the machine's starting state (q_0), then by the input string ($\omega = \omega_1, \omega_2, \ldots, \omega_n$) and ends with one or more blanks or delimiters. For example:

$$x_{1,1,\#} \wedge x_{1,2,q_0} \wedge x_{1,3,w_1} \wedge \cdots \wedge x_{1,n+2,\#}$$

This sets up the initial configuration of the machine exactly as it should be. If this part of the Boolean formula is not true, then the machine did not start correctly, and the tableau is invalid.

(2) Cell Condition(ϕ_{cell}):

We define this part of the formula to make sure that each cell in the tableau contains exactly one symbol. For every row i and column j, we write:

$$\phi_{cell} = \bigwedge_{1 \le i,j \le n^k} \left(\left(\bigvee_{s \in C} x_{i,j,s} \right) \wedge \left(\bigwedge_{\substack{s,t \in C \\ s \neq t}} (\overline{x_{i,j,s}} \vee \overline{x_{i,j,t}}) \right) \right)$$

The first part $\bigvee_{s \in C} x_{i,j,s}$ ensures that at least one symbol is present in each cell.

The second part $\bigwedge_{s \neq t} (\overline{x_{i,j,s}} \vee \overline{x_{i,j,t}})$ ensures that no two symbols are present in the same cell (at most one symbol).

Together, they enforce that each cell contains exactly one symbol.

(3) Accept Condition (ϕ_{accept}): This condition ensures that at least one cell in the tableau contains the accepting state, q_{accept}, which means the machine successfully accepted the input.

We use a disjunction (logical OR) over all positions $[i, j]$ in the tableau to check if the accepting state appears at least once anywhere in the tableau.

The Boolean expression looks like this:

$$\phi_{\text{accept}} = \bigvee_{1 \le i,j \le n^k} x_{i,j,q_{\text{accept}}}$$

This formula is satisfied if there is at least one cell $[i, j]$ in the entire tableau that contains q_{accept}, which indicates that the machine entered an accepting state at some point during its computation.

Move Condition (ϕ_{move}): This condition ensures that the tableau represents a valid step-by-step computation of the Turing machine, M. Specifically, it verifies that each row in the tableau legally follows from the one above it according to the machine's transition function.

We define a "window" as a 2×3 block in the tableau. This window captures a small local region, which is three adjacent cells on one row and the corresponding three cells directly below them on the next row. Since the Turing machine updates only one cell at a time (the one under its read/write head), and the head can move left or right by one cell, this window is large enough to represent the effects of a single transition. Each window compares part of one row (configuration) to the corresponding part in the next row.

Only the cells involved in the head's movement and update may change between rows as only 3 kinds of cells change (The cell that was just written to, the previous head location, and the new head location). All other parts of the row remain unchanged.

Depending on the transition rule applied by M, the content of a valid window must match one of two patterns:

- If $\delta(q, b) \ni (q', c, R)$, meaning the machine reads symbol b in state q, writes c, moves right, and enters state q', then the window looks like:

$$\begin{array}{ccc} a & q & b \\ a & c & q' \end{array}$$

- If $\delta(q, b) \ni (q', c, L)$, meaning the machine reads symbol b in state q, writes c, moves left, and enters state q', then the window looks like:

$$\begin{array}{ccc} a & q & b \\ q' & c & b \end{array}$$

We now define ϕ_{move} to enforce these patterns. Let the six cells in a window with upper-left corner at position $[i, j]$ be:

$$a_1 = x_{i,j,s_1},\ a_2 = x_{i,j+1,s_2},\ a_3 = x_{i,j+2,s_3},\ a_4 = x_{i+1,j,s_4},\ a_5 = x_{i+1,j+1,s_5},\ a_6 = x_{i+1,j+2,s_6}$$

A window is valid if and only if the six symbols $s_1, \ldots, s_6$ represent a legal transition as specified by the transition function δ of M. If it doesn't match, then that part of the tableau is invalid.

We write the move condition as:

$$\phi_{\text{move}} = \bigwedge_{1 \le i,j \le n^k} \left(\bigvee_{\text{legal 6-tuples } (s_1,\ldots,s_6)} (x_{i,j,s_1} \wedge x_{i,j+1,s_2} \wedge x_{i,j+2,s_3} \wedge x_{i+1,j,s_4} \wedge x_{i+1,j+1,s_5} \wedge x_{i+1,j+2,s_6}) \right)$$

This means that for every position $[i, j]$ of a window in the tableau, we want something to be true. Out of all the legal ways a window can look, at least one must match the actual symbols in the window

$$(x_{i,j,s_1} \wedge x_{i,j+1,s_2} \wedge x_{i,j+2,s_3} \wedge x_{i+1,j,s_4} \wedge x_{i+1,j+1,s_5} \wedge x_{i+1,j+2,s_6})$$

. This ensures that the tableau encodes a sequence of steps that the machine could actually take.

Efficiency of the Reduction: Now that we have constructed the Boolean formula

$$\phi = \phi_{\text{start}} \wedge \phi_{\text{cell}} \wedge \phi_{\text{accept}} \wedge \phi_{\text{move}},$$

we must show that the size of this formula and the time it takes to build it is polynomial in the length of the input ω.

First, recall that our tableau is a grid of size $n^k \times n^k$, which means there are n^{2k} cells. For each cell, we have a Boolean variable $x_{i,j,s}$ for every symbol $s \in C$. Since $|C|$ is constant (it includes a fixed number of tape symbols and machine states), we use $O(n^{2k})$ variables overall.

Each subformula contributes the following:

- ϕ_{start} has $O(n^k)$ size, since it describes only the first row of the tableau.
- ϕ_{cell} and ϕ_{accept} each involve all cells, so their sizes are $O(n^{2k})$.
- ϕ_{move} checks all 2×3 windows in the tableau. There are $O(n^{2k})$ such windows, and for each, we check a constant number of valid local patterns. So its total size is also $O(n^{2k})$.

There is an extra $\log(n)$ factor from encoding indices, but even with that, the total size of ϕ is $O(n^{2k} \log n)$, which is still polynomial in n.

Therefore, the entire construction can be done in polynomial time. This confirms that the reduction from any language $A \in NP$ to SAT is polynomial-time.

Since we have shown that:

- $SAT \in NP$,
- every problem in NP is reducible to SAT in polynomial time,

it follows that SAT is NP-complete.

■

Theorem 2: 3-SAT is NP-Complete.

Proof. To prove that $3 - SAT$ is NP-complete, we must show two things:
1. 3-SAT is in NP
2. 3-SAT is NP-hard
$3 - SAT$ is a special version of SAT where each clause has exactly three literals, like $(x_1 \vee \neg x_2 \vee x_3)$.
Example of a $3 - SAT$ formula:

$$(x_1 \vee x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_4 \vee x_5)$$

First, we show that 3-SAT is in NP.
Given a Boolean formula in 3-CNF form (where each clause has exactly 3 literals, as the above example), we can guess a truth assignment to all variables. Then, we check whether this assignment satisfies all clauses.

This checking can be done in polynomial time with respect to the number of variables and clauses. There are polynomial number of clauses, so verification is polynomial. So, 3-SAT is

in NP.

Now, we prove $3-SAT$ is NP-hard.
We already know that SAT is NP-complete (by the Cook-Levin Theorem). So, we reduce SAT to $3-SAT$ to show that $3-SAT$ is at least as hard as SAT.

Let ϕ be a Boolean formula in CNF, which may have clauses with more or less than 3 literals. We now convert ϕ into an equivalent Boolean formula ϕ' in 3-CNF (has exactly 3 literals) such that ϕ is satisfiable if and only if ϕ' is satisfiable.

- If a clause has one literal, say (a), we replace it with $(a \vee a \vee a)$. This does not change the meaning and now it has 3 literals.
- If a clause has two literals, say $(a \vee b)$, we replace it with $(a \vee b \vee b)$. This still keeps it satisfiable because it is converted to 3-CNF.
- If a clause has three literals, we leave it as is because it is already in 3-CNF form.
- If a clause has more than three literals, say 5 literals, $a_1 \vee a_2 \vee a_3 \vee a_4 \vee a_5$ (General formula: $a_1 \vee a_2 \vee \cdots \vee a_k$, where $k > 3$), we introduce new variables, let's say z_1 and z_2 (General formula: $z_1, z_2, \ldots, z_{k-3}$). We break up the big clause into several 3-literal clauses using these new variables.

  $$(a_1 \vee a_2 \vee z_1) \wedge (\neg z_1 \vee a_3 \vee z_2) \wedge (\neg z_2 \vee a_4 \vee a_5)$$

  This transformation maintains satisfiability, because if the original long clause is satisfiable, one of these short clauses will be satisfied. If these short clauses are satisfiable, that means that one of the original literals is true and thus, ϕ is true.

This transformation keeps the satisfiability of the formula the same. That is, the new formula, ϕ', is satisfiable if and only if the original formula, ϕ, was. The size of the new formula grows only by a constant factor and can be computed in polynomial time.

Conclusion:
Since 3-SAT is in NP and any instance of SAT can be reduced to an instance of 3-SAT in polynomial time, it follows that 3-SAT is NP-complete. ■

4. Grid Games NP-Completeness

4.1. **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.

This problem has been shown to be NP-complete [JH19].
To be NP-complete,
1. It must be in NP
2. It must be NP-hard
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 $\in NP$.
To prove that Latin Square Completion is NP-hard, [JH19] uses reduction from memetic graph coloring. This is beyond the scope of this paper. So, we just consider the result that LSC is NP-hard.

Conclusion:
Since Latin Square Completion is both in NP and is NP-hard, Latin Square Completion is NP-complete.

Proving Sudoku is NP-complete.

Proof. This proof follows from [Hoe20].
To show that Sudoku is NP-complete, we must prove two things:
1. Sudoku is in NP
2. Sudoku is NP-hard

First, we show that Sudoku $\in NP$.
We consider the generalized Sudoku problem on an $n^2 \times n^2$ grid, where each cell must contain a digit from $\{1, 2, \ldots, n^2\}$.
The constraints for the numbers in each cell are:

- Each row must contain all digits from 1 to n^2 exactly once.
- Each column must contain all digits from 1 to n^2 exactly once.
- Each of the $n \times n$ subgrids (blocks) must contain all digits from 1 to n^2 exactly once.
- Some cells are pre-filled as part of the input puzzle.

Given a candidate filled grid (a certificate), 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, we show that Sudoku is NP-hard.

We prove this by reduction from the Latin Square Completion (LSC) problem, which is known to be NP-complete (as proven above).

Recall that 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. It is similar to a Sudoku, except that it does not require the subgrids (blocks) to contain all digits from 1 to n^2 exactly once. The Latin Square Completion problem asks whether a partially filled Latin square can be completed to a full Latin square.

Given an instance of the 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 of size $n^2 \times n^2$. The idea is to embed the Latin square inside the Sudoku grid. 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 satisfies all conditions of uniqueness: rows, columns and subgrids (blocks).

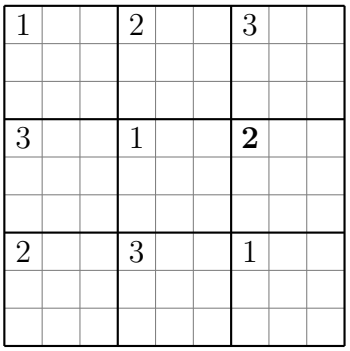
- For each known entry $L(i, j) = a$, we fix the corresponding cell in the Sudoku grid:

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

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.

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



Latin Square (3×3) **Sudoku Grid (9×9)**

These figures show a sample embedding from a Latin Square to a Sudoku. Note that even though real Sudoku puzzles usually need more clues, here we only embed nine. This is just enough to preserve the logic of the Latin square. We only need to check if a valid solution exists. We do not need to find a unique one.

- The remaining entries of the Sudoku are left blank. The rules of Sudoku enforce Latin square properties via:
  - Row uniqueness,
  - Column uniqueness,
  - Block uniqueness.

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

$$\text{The Latin square } L \text{ is completable} \iff \text{The Sudoku instance } S \text{ is solvable}$$

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

Conclusion:
Since

- A proposed solution to a Sudoku puzzle can be verified in polynomial time, and

- Solving Sudoku is at least as hard as solving the NP-complete Latin Square Completion problem via a polynomial-time reduction,

Sudoku is NP-complete.

■

5. $PSPACE$-completeness

In this section, we will be looking at a proof of a well-known problem, which is proven to be $PSPACE$ complete. This proof follows from [AB09]. From this proof we understand that $TQBF$ corresponds to $PSPACE$ in a similar way by which SAT corresponds to NP. First, we understand what $TQBF$ is.
$TQBF$ stands for True Quantified Boolean Formula. A sample would look like this:

$$\exists x_1 \forall x_2 \exists x_3 \ldots \varphi(x_1, x_2, x_3, \ldots)$$

where φ is a Boolean Formula (like SAT) and the quantifiers alternate.

Theorem: $TQBF$ is $PSPACE$-complete.

Proof. To prove this, we need to show two things:
1. $TQBF$ is in $PSPACE$, and
2. $TQBF$ is $PSPACE$-Hard

First, we show that $TQBF$ is in $PSPACE$.
Let $\Psi = Q_1 x_1 \, Q_2 x_2 \, \ldots Q_n x_n \, \varphi(x_1, \ldots, x_n)$ be a fully quantified Boolean formula, where each $Q_i \in \{\forall, \exists\}$.
Here, Ψ is an expression made of quantifiers $(\exists, \forall)$, variables $(x_1, x_2, \ldots)$, and a Boolean formula φ. Also, φ is the Boolean formula that we trying to make true.

We can decide the truth of Ψ using a recursive procedure:

- If there are no remaining quantifiers, evaluate the propositional formula φ. Once all the quantifiers are gone, evaluate φ like a SAT instance. We just plug in the values of the variables and check if the formula is true.
- If the first quantifier is $\exists x_i$, then check if Ψ is true for $x_i = 0$ or $x_i = 1$. Here, we check if at least one of the two assignments (0 or 1) makes the rest of the formula true.
- If the first quantifier is $\forall x_i$, then check if Ψ is true for both $x_i = 0$ and $x_i = 1$. here, we check if both values make the formula true because it must be true for all possible values.

At each step, we only need to remember the current variable assignment (which takes $O(n)$ space) and the formula φ, so the space used is polynomial in the size of Ψ. So, even

though there are exponentially many branches, we use only polynomial space to track each one. Therefore, $TQBF \in PSPACE$.

Now, we prove that $TQBF$ is $PSPACE$-Hard.
Let $L \in PSPACE$, where L is any problem. Then there exists a deterministic Turing machine M that decides (solves) L using at most $S(n)$ space for inputs of length n, where $S(n)$ is a polynomial. This means that M can solve the problem in polynomial space.

Let w be an input of length n. We construct a quantified Boolean formula Ψ_w such that:

$$\Psi_w \text{ is true} \iff M \text{ accepts } w$$

In simpler words, we are trying to reduce any $PSPACE$ problem to $TQBF$. We build a formula Ψ_w that is true if and only if M accepts w.

Let C_{start} and C_{accept} be the start and accepting configurations of M on w. A configuration is a snapshot of M's current state, tape content, and head position. Each configuration uses $O(S(n))$ space because this is the tape's bound. So we can encode a configuration as a binary string of polynomial length.

There are at most $T = 2^{O(S(n))}$ possible configurations (since space is limited), so the maximum number of steps in the computation is also at most T. This is justified because if the machine only has $2^{S(n)}$ possible configurations, then that longest computation (before accepting or repeating) must be $\leq$ than that.
To clarify further, since a Turing machine using at most $S(n)$ space is limited to writing on a polynomial-sized portion of its tape and has a finite number of states and tape symbols, the total number of possible configurations is at most $2^{O(S(n))}$, which is exponential in the space used. This bound ensures that the cannot run for more than an exponential number of steps without repeating a configuration, which would imply an infinite loop. However, this does not mean that the machine is using exponential space. It still uses only $S(n)$, which is polynomial. The exponential bound here refers to time and not space. When we simulate this machine using a quantified Boolean formula in the $TQBF$ reduction, we construct the formula in a way where each recursive call keeps track of a constant number of configurations and time bounds. Importantly, this simulation only requires polynomial space to store intermediate configurations and quantifiers at each level. Hence, although the number of steps may be exponential, the simulation remains within PSPACE.

We define a recursive Boolean formula $\text{PATH}(C_1, C_2, t)$ that is true if and only if M can go from configuration C_1 to C_2 in t or fewer steps. The idea is to divide the path into two halves:

$$\text{PATH}(C_1, C_2, t) = \exists M \forall (C_i, C_j) \in \{(C_1, M), (M, C_2)\} \text{ PATH}(C_i, C_j, t/2)$$

This is done so that we simulate the entire computation of M on w by checking whether a valid midpoint configuration exists.

This reduces the time bound t by half in each recursive step. This leads to a recursion depth of $logT = O(S(n))$. So, the resulting Boolean formula will be of polynomial size.

Hence, we construct a fully quantified Boolean formula Ψ_w such that:

$$\Psi_w = \text{PATH}(C_{\text{start}}, C_{\text{accept}}, T)$$

and $\Psi_w \in TQBF$ is true if and only if M accepts w.

Conclusion:

We have shown:

1. $TQBF \in PSPACE$
2. $TQBF$ is $PSPACE$-hard

Therefore, $TQBF$ is $PSPACE$-complete.

■

6. Formula-Game

In this section we explore winning strategies for games through the Formula-Game, which is a very interesting yet simple way to look at PSPACE problems [AB09].

Formula-Game explanation. Let $\phi = Q_1 x_1 \, Q_2 x_2 \, Q_3 x_3 \ldots Q_k x_k \; [\psi]$ be a fully quantified Boolean formula, where each Q_k is either an $\exists$ or $\forall$ quantifier , and ψ is the Boolean formula with no quantifiers.

This can be represented as grame between 2 players:

(1) Two players, Player A and Player B, take turns assigning truth values to the variables $x_1, x_2, \ldots, x_k$ in the order they appear.
(2) Player A chooses values for all variables bound by the universal quantifier $\forall$, and Player B chooses values for all variables bound by the existential quantifier $\exists$.
(3) Both Players choose one at a time. For example, if the order of quantifiers is $\exists \; \forall \; \exists$, Player B chooses first, Player A goes second and then Player B chooses the third value.
(4) Once all variables are assigned, we evaluate the Boolean formula ψ under those assignments. If ψ is true, Player B wins the game. If ψ is false, Player A wins.

Let's consider an example:

$$\phi_1 = \exists x_1 \forall x_2 \exists x_3 \, [(x_1 \vee x_2) \wedge (x_2 \vee x_3) \wedge (\neg x_2 \vee \neg x_3)]$$

In this formula, the quantifiers alternate between $\forall$ and $\exists$, so:

- Player A chooses x_2 (the $\forall$ quantified variables).
- Player B chooses x_1 and x_3 (the $\exists$ quantified variable).
- Note that Player B chooses x_1, then Player A chooses x_2, and finally, Player B chooses x_3.

Now, we simulate a round of the game:

- Suppose Player B chooses $x_1 = 1$.
- Player A chooses $x_2 = 0$.
- Player B chooses $x_3 = 1$.

Substituting these into ϕ_1, we get:

$$(1 \vee 0) \wedge (0 \vee 1) \wedge (\neg 0 \vee \neg 1) = 1 \wedge 1 \wedge 1 = 1$$

Since the formula evaluates to 1, which means true, Player B wins. In fact, Player B has a winning strategy in this game. If Player B always chooses $x_1 = 1$, and then selects x_3 to be the negation of whatever Player A chooses for x_2, it is guaranteed that ϕ_1 evaluates to true no matter what Player A chooses for x_2.

Next, consider a formula where Player A has a winning strategy:

$$\phi_2 = \exists x_1 \forall x_2 \exists x_3 \left[(x_1 \vee x_2) \wedge (x_2 \vee x_3) \wedge (x_2 \vee \neg x_3)\right]$$

The quantifier structure is the same, so Player A again chooses x_2 and Player B chooses x_1 and x_3. Player A now has a winning strategy because no matter what Player B selects for x_1, Player A may select $x_2 = 0$, thereby falsifying the part of the formula appearing after the quantifiers. This is not affected by Player B's last move.

Now, we prove that Formula-Game is $PSPACE$-complete.

Theorem: Formula-Game is $PSPACE$-complete.

Proof. The formula $\phi = \exists x_1 \forall x_2 \exists x_3 \ldots [\psi]$ is True when there exists a case of x_1 such that, for any case of x_2, a case of x_3 exists such that, and so on $\ldots$, where ψ is True under those cases of the variables. Similarly, Player B has a winning strategy in the game when Player B can make some assignment to x_1, such that for any value of x_2, Player B can make an assignment to x_3 such that, and so on $\ldots$, ψ is True under these setting of the variables.
The same reasoning applies when the formula does not alternate between $\exists$ and $\forall$ quantifiers. If $\psi = \forall x_1, x_2, x_3 \exists x_4, x_5 \forall x_6 [\psi]$, Player A would make the first three moves as it chooses for $\forall$ and assigns values to x_1, x_2, x_3. Then, Player B would make 2 moves to assign values to x_4 and x_5. Finally, Player A would assign a value to x_6.

Hence, $\phi \in TQBF$ exactly when $\phi \in$ Formula-Game.
Thus, Formula-Game is $PSPACE$-complete because it is the same as $TQBF$

$$Formula - Game = TQBF$$

■

7. Another Sudoku Relation

Earlier, we proved that Sudoku is NP-complete.
Now, we investigate whether Sudoku is $PSPACE$-complete.

Recall that for a problem to be $PSPACE$-complete:

(1) It must be in $PSPACE$, and
(2) It must be $PSPACE$-hard—i.e., at least as hard as any other problem in $PSPACE$.

1. Sudoku is in $PSPACE$

This follows from the known complexity class hierarchy:

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

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

2. Is Sudoku $PSPACE$-hard?

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 $TQBF$.

Examples include:

- $TQBF$ (True Quantified Boolean Formula),
- The Formula Game,
- Generalized Geography.

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 $NP \neq PSPACE$.

Analogy for Intuition

Problem Type	Analogy	Examples
NP-complete	Solving a puzzle in one go	Sudoku, SAT
$PSPACE$-complete	Playing a game with turns	$TQBF$, Formula Game

Conclusion. Sudoku is NP-complete and hence in $PSPACE$, but we have no known reduction proving it to be $PSPACE$-hard. Therefore, Sudoku is not currently known to be $PSPACE$-complete, and most evidence suggests it likely is not, unless $NP = PSPACE$, which remains an open problem.

8. Open-problems and other relationships between complexity classes

In this section, we highlight some of the most important open questions in computational complexity theory. These problems are central to understanding how various complexity classes such as P, NP, $PSPACE$, and the Polynomial Hierarchy (PH) are related, [Ueh11].

1. The P vs NP Problem. The most famous open problem in complexity theory is:

$$\text{Does } P = NP?$$

This asks whether every problem whose solution can be verified quickly (in polynomial time) can also be solved quickly. If $P = NP$, it would mean that all NP-complete problems such as SAT, 3-SAT, and Sudoku can be solved in polynomial time.

Currently, no one knows the answer. Most complexity theorists believe that $P \neq NP$.

A proof, which could prove either of the above ($P = NP$ or $P \neq NP$) would have enormous consequences across mathematics and computer science.

2. NP vs PSPACE. Another major open question is:

$$\text{Does } NP = PSPACE?$$

We know that:

$$P \subseteq NP \subseteq PSPACE$$

But whether these inclusions are strict is not known. Most researchers believe:

$$P \subset NP \subset PSPACE$$

This means that $PSPACE$-complete problems (such as $TQBF$ or Formula-Game) are believed to be strictly harder than NP-complete problems.

3. PSPACE and EXPTIME. To better understand the relationship between $PSPACE$ and $EXPTIME$, we first define the class $EXPTIME$.

Definition: EXPTIME. The class $EXPTIME$ (Exponential Time) is the set of all decision problems that can be solved by a deterministic Turing machine in exponential time. Formally,

$$EXPTIME = \bigcup_{k \in \mathbb{N}} \text{TIME}(2^{n^k})$$

This means that a problem is in $EXPTIME$ if it can be solved in time bounded by an exponential function of the input size.

Example Problems in $EXPTIME$:

- Generalized Chess: Determining whether a player has a winning strategy in an $n \times n$ chessboard is in $EXPTIME$.
- Alternating Turing Machine Acceptance: For certain machines and bounds, determining acceptance can use exponential time.

Now, we look at the relationship with $PSPACE$.

We have the known inclusion:

$$PSPACE \subseteq EXPTIME$$

If a problem can be solved using polynomial space, it can be simulated using exponential time, because the number of possible configurations of a polynomial-space Turing machine is at most exponential. Why $PSPACE \subseteq EXPTIME$?

Suppose a Turing machine uses at most polynomial space (like n^k cells) on an input of size n. Even if it runs for a long time, the number of different configurations (tape contents, head position, and state) it can be in is limited.

Since each configuration only uses polynomial space, the total number of different configurations is at most exponential in n.

We can simulate the machine by:

- Listing all possible configurations, which are exponentially many.
- Checking if the machine can go from the starting configuration to the accepting one.

This takes exponential time, but is always possible. So every problem that can be solved in polynomial space can also be solved in exponential time. Therefore,

$$PSPACE \subseteq EXPTIME$$

The Separation Between $PSPACE$ and $EXPTIME$ has been proven:

$$PSPACE \neq EXPTIME$$

This separation follows from a time-space hierarchy theorem, which shows that problems solvable in exponential time strictly contain those solvable in polynomial space. This means that there are some problems that can be solved in exponential time, but not with only

polynomial space. In other words, some problems in $EXPTIME$ are too big to fit within the memory limits of $PSPACE$.
Why this matters:

- Some problems need a lot of time and a lot of memory to solve.
- Allowing a machine to run for a long time (here exponential time) does not mean it can solve problems using just a small amount of memory.
- $PSPACE$ problems are limited by memory, while $EXPTIME$ problems are limited by time.

This is one of the few clear separations between major complexity classes. It tells us that giving a computer more time (exponentially more) lets it solve problems that just can't be done with limited memory alone. So, $EXPTIME$ is more powerful than $PSPACE$.

4. The Polynomial Hierarchy (PH). The Polynomial Hierarchy (PH) is a generalization of the classes P, NP, and $coNP$ using alternating quantifiers. It contains levels such as $\Sigma_1^P = NP$ and $\Pi_1^P = coNP$, and higher levels with alternating existential and universal quantifiers.

Definition of $coNP$: The class $coNP$ consists of the complements of problems in NP. That is, a language L is in $coNP$ if and only if its complement $\overline{L}$ is in NP. In simpler words, while NP is the class of problems for which a "yes" answer can be verified quickly (in polynomial time), $coNP$ is the class of problems for which a "no" answer can be verified quickly.
Example of a problem in $coNP$: Consider the problem of checking whether a Boolean formula is a tautology, that is it is always true. This problem is in $coNP$, because its complement, which is checking whether a formula is not a tautology (it has at least one false assignment), is in NP.

A big question in complexity theory is does the Polynomial Hierarchy collapse? An important open question is whether the hierarchy is strict at each level. If $NP = coNP$ or $NP = P$, the entire hierarchy would collapse to a lower level.

Importance of solving these problems. These open problems affect fields such as cryptography. Most modern encryption relies on the assumption that $P \neq NP$. Cryptography deals with hard and important tasks like protecting passwords, private messages, bank accounts, and national security. It relies on very difficult mathematical problems that current computers are not able to solve quickly. Problems like factoring large numbers or solving complex puzzles with no known shortcuts form the core of secure encryption. Complexity

theory helps us understand how difficult these problems are by grouping them into classes like P and NP. The more difficult the problem, the more secure the system.
The challenge is that we still do not know for sure whether these problems truly require a lot of time to solve. So far, no one has discovered an efficient solution, but that does not mean one cannot exist. If open questions such as "Is $P = NP$?"are ever answered, it could completely change the way cryptography and digital security work.

Relationships between complexity classes. We currently have the following known inclusions between classes, [Ueh11]:

$$L \subseteq NL \subseteq P \subseteq NP \subseteq PSPACE \subseteq NPSPACE \subseteq EXP \subseteq NEXP$$

The definitions and purposes of the other classes (not mentioned in this paper) can be explored in [AB09, Sip12, Ueh11]

Ultimately, complexity theory helps us understand not just what computers can do, but why some problems remain inherently hard to solve, and how answering these questions could reshape computation, cryptography, and beyond.

References

[AB09] Sanjeev Arora and Boaz Barak. *Computational Complexity: A Modern Approach*. Cambridge University Press, 2009.

[FH03] Lance Fortnow and Steven Homer. A short history of computational complexity. *Bulletin of the EATCS*, 80:95–133, 2003.

[Hoe20] Eline S. Hoexum. Sudoku and computational complexity. Bachelor's thesis, University of Groningen, 2020.

[JH19] Yan Jin and Jin-Kao Hao. Solving the latin square completion problem by memetic graph coloring. *IEEE Transactions on Evolutionary Computation*, 23(6):1015–1028, 2019.

[Sip12] Michael Sipser. *Introduction to the Theory of Computation*. Cengage Learning, third edition, 2012.

[Tri24] Eliza L. Tripp. The cook-levin theorem: Lecture notes. Department of Computer Science, Bucknell University, 2024.

[Ueh11] Ryuhei Uehara. Computational complexity of puzzles and related topics. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, E94-A(6):1319–1325, 2011. Invited Paper.