

Reduction in Theory of Computation

IngramMaths

January 2026

1 Introduction

Definition 1.1 (Decidability)

Suppose L is a language over an alphabet Σ . We say that L is **decided** by a Turing machine M if and only if M accepts L and M halts on every input. We say that a language is **decidable** if and only if there exists a Turing machine which decides it.

Definition 1.2 (Time Complexity)

Suppose we have a Turing machine M which halts on every input. Then, we define the **time complexity** of M to be a function $\tau_M : \mathbb{N} \rightarrow \mathbb{N}$ such that $\tau_M(n)$ is the maximum number of transitions needed for M to halt on a string of length n .

Remark. Time complexity is defined analogously for nondeterministic Turing machines: this is just the maximum number of transitions *on a given branch*.

Definition 1.3 (Big-O Notation)

Suppose we have two functions $f, g : \mathbb{N} \rightarrow \mathbb{N}$. We say that f is of (worst-case) **order** g if and only if there exist $n_0, c \in \mathbb{N}$ such that

$$\forall n \in \mathbb{N} \quad n \geq n_0 \implies f(n) \leq cg(n)$$

We define $O(g)$ (often informally written $O(g(n))$) to be the set of all functions $\mathbb{N} \rightarrow \mathbb{N}$ of order g .

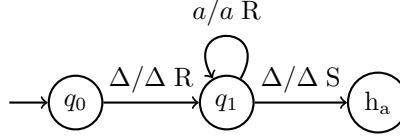
Definition 1.4 (Time Complexity Classes)

Suppose L is a language over an alphabet Σ . We say that L is decidable in **polynomial time** (or **tractable**) if and only if there exists a (deterministic) Turing machine M which decides L such that $\tau_M \in O(n^k)$ for some $k \in \mathbb{N}$. We call the class of all such languages P .

We say that L is decidable in **nondeterministic polynomial time** if and only if there exists a nondeterministic Turing machine M which decides L such that $\tau_M \in O(n^k)$ for some $k \in \mathbb{N}$. We call the class of all such languages NP .

Example 1.5

We would expect $L = \{a^n \in \{a\}^* \mid n \in \mathbb{N}\}$ to be tractable, that is, to belong to P. To prove this formally, we need to outline a construction for a TM which decides it, and then prove that its time complexity is polynomial carefully. Below is such a Turing machine, with accepting state h_a :



This Turing machine clearly accepts L and halts on every input. As for its time complexity, the worst-case number of transitions for a string of length $n \in \mathbb{N}$ would be

$$1 + n + 1 = (n + 2)$$

So, we have $\tau_M : \mathbb{N} \rightarrow \mathbb{N}$ defined by

$$\tau_M(n) = n + 2$$

Ordinarily, we would not often go through the formal steps of proving that this falls in a polynomial class, but we will do so here for completeness without relying on unstated theorems. Picking $n_0 = 1$ and $c = 3$, if $n \geq n_0 = 1$ then

$$n + 2 \leq n + 2n = 3n = cn$$

It follows that $\tau_M \in O(n)$. Hence, indeed $L \in P$.

Theorem 1.6

Suppose that $L \in P$. Then, $L \in NP$. That is, $P \subseteq NP$.

Proof. Since $L \in P$, there exists a Turing machine $M = (Q, \Sigma, \Gamma, q_0, \delta)$ accepting L such that $\tau_M \in O(n^k)$ for some $k \in \mathbb{N}$. We hence construct from this a nondeterministic Turing machine

$$N = (Q, \Sigma, \Gamma, q_0, \rho)$$

with a new **transition relation** ρ given by

$$\rho = \{(q, \sigma, q', \sigma', d) \mid (q', \sigma', d) = \delta(q, \sigma)\} \subseteq 2^{Q \times \Gamma \times Q \times \Gamma \times \{L, S, R\}}$$

This relation is equivalent to the transition function of M (in fact, they are literally equal to each other set-theoretically) and therefore both $L(N) = L(M) = L$, and $\tau_M = \tau_N$. Thus, indeed L is accepted by an NTM in polynomial time, that is, $L \in NP$. \square

Definition 1.7 (Polynomial-Time Reduction)

Suppose L_1, L_2 are languages over an alphabet Σ . A **reduction** from L_1 to L_2 is a (total) computable function $f : \Sigma^* \rightarrow \Sigma^*$ such that

$$x \in L_1 \iff f(x) \in L_2$$

A **polynomial-time reduction** is a reduction which is computable by a Turing machine M such that $\tau_M \in O(n^k)$ for some $k \in \mathbb{N}$.

Theorem 1.8

Suppose that L_1 reduces to L_2 . Then,

- (i) If L_2 is decidable, then L_1 is decidable.
- (ii) If $L_2 \in P$ and the reduction is in polynomial time, then $L_1 \in P$.
- (iii) If $L_2 \in NP$ and the reduction is in polynomial time, then $L_1 \in NP$.

Proof. (i). Since L_1 reduces to L_2 , there exists a computable function $f : \Sigma^* \rightarrow \Sigma^*$ such that

$$x \in L_1 \iff f(x) \in L_2$$

Since f is computable, there exists a Turing machine M_f which computes f . Since L_2 is decidable, there exists a Turing machine M_2 which decides L_2 . We hence construct a Turing machine M_1 which first runs M_f on a string x , and hence runs M_2 on $f(x)$. We won't formally show here that this is possible: it can be done very easily by assuming without loss of generality that the two TMs have distinct non-halting states, and replacing the accepting state of M_f with the initial state of M_2 .

If $x \in L_1$ then $f(x) \in L_2$. Hence, $f(x)$ is accepted by M_2 and thus x is accepted by M_1 . Conversely, if $x \notin L_1$ (but $x \in \Sigma^*$) then $f(x) \notin L_2$. Hence, $f(x)$ is not accepted by M_2 , and therefore x is not accepted by M_1 . Thus,

$$L(M_1) = L_1$$

We know that M_1 halts on every input, since M_f takes a finite number of transitions to transform $x \mapsto f(x)$ and M_2 halts on every input. Hence, M_1 decides L_1 , that is, L_1 is decidable.

(ii). Suppose that, additionally to (i), $L_2 \in P$ and f is a polynomial-time reduction. Then, we may assume M_f and M_2 from (i) both have polynomial time complexities without loss of generality, say

$$\tau_{M_f} \in O(n^s), \quad \tau_{M_2} \in O(n^t)$$

for some $s, t \in \mathbb{N}$. It follows that

$$\tau_{M_1} \in O(n^s + n^t) = O(n^{\max\{s, t\}})$$

It follows that $L_1 \in P$.

(iii). Alternatively to (ii), suppose that $L_2 \in NP$ and f is a polynomial-time reduction. Then, we may assume M_f is a TM with a polynomial time complexity and M_2 is a nondeterministic Turing machine with a polynomial time complexity without loss of generality.

In this case, the proof is analogous to (ii), but with M_1 being a nondeterministic Turing machine, giving $L_1 \in \text{NP}$ as required. \square

Definition 1.9 (Decision Problems)

A **decision problem** can be seen logically as a predicate for which we want to determine truth or falsity for given inputs. That is, given some inputs, we want to determine whether some property of those inputs holds.

An **instance** of a decision problem refers to the evaluation of the predicate for a given input: a "yes-instance" is one which evaluates true, and a "no-instance" is one which evaluates false.

Remark. When studying decision problems in computability theory, we associate them with some kind of string encoding such that a Turing machine is able to interpret them, trying to find the language of all encoded yes-instances. For example, consider the following problem:

Given a pair of numbers $m, n \in \mathbb{N}$, is $m > n$?

This problem could be associated with an encoding of the form $1^m 0 1^n$, for example. We will assume without formal justification that there exists a valid encoding for Turing machines: we will encode M as $e(M)$.

Example 1.10

It is known that the halting problem (HP) is undecidable: this is the problem of determining whether, given a pair (M, w) consisting of a Turing machine and a string over its input alphabet, M halts when processing w . We will now use this to show that the membership problem (MP) for Turing machines is undecidable.

The membership problem is the problem of determining whether, given a pair (M, w) , we have $w \in L(M)$. Here, we will not go into enough detail to be concerned about the nature of the encoding. In order to show that MP is undecidable, we can reduce HP to it. Indeed, if MP were decidable and we had a reduction then 1.8 would tell us that HP is decidable, which would be a contradiction.

For this reduction, we construct a function $f : \Sigma^* \rightarrow \Sigma^*$ which maps non-encodings to the empty string Λ and otherwise mapping $e(M, w) \mapsto e(M', w)$. Here, M' is a modification of M for which all transitions to the rejecting state are replaced with transitions to the accepting state. We claim without formal proof that such a mapping is computable.

If $e(M, w)$ is a yes-instance of HP then M and hence M' will halt on w . Since all transitions to a halting state in M' are to the accepting state, it follows that M' accepts w , so $w \in L(M)$. Thus, $e(M', w)$ is a yes-instance of MP.

Conversely, if an input is not a yes-instance of HP then there are two possibilities. One is that the input is not a valid encoding, in which case it will be mapped to Λ and not be a yes-instance of MP. Alternatively, the input could be a non-halting pair $e(M, w)$, in which case M' does not halt on w . Thus, $w \notin L(M')$ in this case.

Combining this all together, we get that x is a yes-instance (encoded) of HP if and only if $f(x)$ is a yes-instance (encoded) of MP. Since we claimed that f is computable, it follows that f reduces HP to MP (actually, it reduces the language of yes-instances, but this is enough as mentioned before). Hence, it follows that MP must be undecidable.

Example 1.11

It is known that the language

$$L_2 = \{ww \in \Sigma^* \mid w \in \Sigma^*\}$$

is in P. We shall now show that therefore the language

$$L_1 = \{w \in \Sigma^* \mid w = w^R\}$$

(where \cdot^R is the reversal of a string) is also in P. To do this, it is enough to construct a polynomial-time reduction from L_1 to L_2 . In particular, define $f : \Sigma^* \rightarrow \Sigma^*$ by

$$f(x) = xx^R$$

If $x \in L_1$ then $x = x^R$. Thus, we get

$$f(x) = xx^R = xx \in L_2$$

Conversely, suppose that for a given $x \in \Sigma^*$, we have $f(x) = xx^R \in L_2$. Then, for some $w \in \Sigma^*$, $xx^R = ww$. Since x and w both make up the first half of this string, it follows that $x = w$ and so $ww^R = ww$. That is, $w = w^R$, and so $x = x^R$. Hence, $x \in L_1$. Thus,

$$x \in L_1 \iff f(x) \in L_2$$

Further, it is known that f is computable: the Turing machine algorithm would roughly work on a string x as follows:

- Starting at the end of x , go through each symbol in x , marking the symbols as they are dealt with.
- For each symbol, travel to the end of the current working string and append it there.
- Then, travel back to find the next unmarked symbol in the string.
- If there is no such unmarked symbol, go to the first cell of the tape and accept.

Hence, f is a reduction from L_1 to L_2 . Further, f is a polynomial-time reduction.

Indeed, the TM computing f would need to traverse at most a multiple of the whole length of the string for each symbol, giving $O(n^2)$ time complexity. Hence, since L_1 has a polynomial-time reduction to L_2 and $L_2 \in P$, it follows that $L_1 \in P$.

2 NP-Completeness

Definition 2.1 (NP-Completeness)

Let L be a language. We say that L is **NP-hard** if and only if every language in NP reduces to L in polynomial time. We say that L is **NP-complete** if and only if it is both in NP and NP-hard.

Theorem 2.2

Suppose L is NP-complete. If $L \in P$, then $P = NP$.

Proof. Suppose that $L \in P$. It is known already that $P \subseteq NP$, so it's enough to show that also $NP \subseteq P$. Take any language $L' \in NP$. Since there exists a polynomial-time reduction from L' to L (as a result of L being NP-hard) and $L \in P$ it follows that $L' \in P$. Since L' was arbitrary, it follows that $NP \subseteq P$. Hence, $P = NP$. \square

Remark. We will shortly look at some examples of NP-complete problems. This theorem establishes, very importantly, that finding a deterministic polynomial-time algorithm for solving any one of them is enough to prove $P = NP$, which is arguably the most famous unsolved problem in computer science.

Theorem 2.3

Suppose that $P = NP$. Then, for all $L \in P \setminus \{\Sigma^*, \emptyset\}$, L is NP-complete.

Proof. It's trivial that all such L are in NP, and hence it's enough to show that L is NP-hard. Take any language $L' \in NP$. Since $P = NP$, it follows that $L' \in P$. Since L is not Σ^* or \emptyset , we can take any fixed $x_L \in L$ and $x_{\bar{L}} \in \bar{L}$. Thus, we construct the following function $f : \Sigma^* \rightarrow \Sigma^*$:

$$f(x) = \begin{cases} x_L & \text{if } x \in L' \\ x_{\bar{L}} & \text{otherwise} \end{cases}$$

It's trivial to see that therefore

$$x \in L' \iff f(x) \in L$$

Since $L' \in P$, we can construct a polynomial-time TM which computes f :

- Check if $x \in L'$ in polynomial time.
- If so, then erase and write x_L to the tape.
- Otherwise, erase and write $x_{\bar{L}}$ to the tape.

Hence, f is a polynomial-time reduction from L' to L . Since L' was arbitrary, it follows that L is NP-hard as required. \square

Definition 2.4 (Satisfiability Problem)

A boolean expression in **CNF** (conjunctive normal form) is one which can be written as

$$\Phi \triangleq \bigwedge_{i=1}^n \bigvee_{j=1}^{m_i} L_{i,j}$$

where each $L_{i,j}$ is a **literal**, that is, either a variable or the negation of a variable. The notation \bigwedge refers to repeated conjunction ("and") and \bigvee refers to repeated disjunction ("or").

A boolean expression is said to be **satisfiable** if and only if there exists an assignment of truth values to its variables such that it holds true.

The **satisfiability problem** (Sat) is as follows:

Given a boolean expression Φ in CNF, determine whether Φ is satisfiable.

Example 2.5

Let's determine whether the following CNF boolean expressions are satisfiable:

- (i) $(P \vee Q) \wedge (P \vee \neg Q)$
- (ii) $(P \vee Q) \wedge (\neg P \vee Q) \wedge \neg Q$

For (i), we can make the assignment $\{P \mapsto \text{true}, Q \mapsto \text{false}\}$. This causes the expression to evaluate true, meaning that it is satisfiable.

For (ii), we see that Q must evaluate false due to the final conjunct $\neg Q$. Hence, to ensure that all conjuncts are true, since Q is false, we would need both P and $\neg P$ to be true simultaneously. This is not possible (we say P and $\neg P$ are **complementary**) and therefore the expression is not satisfiable.

Theorem 2.6 (Cook-Levin Theorem)

The Sat problem is NP-complete.

Remark. We won't prove this result here, as the proof is quite extensive. Showing that Sat is in NP is not excessively difficult due to us being able to use a guess-and-check strategy with an NTM. Showing that Sat is NP-hard can be done by representing the execution of an NTM in a tableau, and modelling the operations on this tableau in boolean logic. This is possible because the polynomial time complexity (for nondeterministic TMs) bounds the space complexity, giving a finite tableau and a corresponding boolean expression which can be computed in polynomial time.

Theorem 2.7

Suppose that L_1 is NP-hard, and that L_1 reduces to L_2 in polynomial time. Then, L_2 is NP-hard.

Proof. Take an arbitrary $L \in \text{NP}$. By the fact that L_1 is NP-hard, there exists a reduction $f_1 : \Sigma^* \rightarrow \Sigma^*$ such that

$$x \in L \iff f_1(x) \in L_1$$

Similarly, since L_1 reduces to L_2 , there exists a reduction $f_2 : \Sigma^* \rightarrow \Sigma^*$ such that

$$y \in L_1 \iff f_2(y) \in L_2$$

It follows that

$$x \in L \iff f_2(f_1(x)) \in L_2$$

We may assume without loss of generality that f_1 and f_2 are computable in polynomial time. It follows that their composition is computable in polynomial time (just replace the accepting state of a TM computing f_1 with the start state of f_2 , assuming without loss of generality that these TMs have distinct non-halting states). Hence, $f_2 \circ f_1$ is a polynomial-time reduction from L to L_2 . Since L was arbitrary, it follows that L_2 is NP-hard. \square

Theorem 2.8

The 3-Sat problem, stated as follows, is NP-complete:

Given a boolean expression Φ in CNF with exactly 3 literals in each conjunct, determine whether Φ is satisfiable.

Proof. The proof that 3-Sat is in NP is effectively identical to that for Sat. Indeed, given an expression

$$\Phi \triangleq \bigwedge_{i=1}^n (L_{i,1} \vee L_{i,2} \vee L_{i,3})$$

we can nondeterministically select a valid configuration of truth values for the different truth variables, and then check whether the resulting evaluation of Φ is true in polynomial time, accepting if so. As for showing that 3-Sat is NP-hard, we need to use the previous theorem.

To reduce Sat to 3-Sat, all we need to do is transform an expression Φ in CNF into an expression Ψ in CNF with at most three literals for each conjunct. If Φ has a conjunct with less than three literals, we can just add an arbitrary literal or two to it as necessary. So now suppose that Φ has a conjunct with too many literals, say

$$D \triangleq \bigvee_{i=1}^n L_i$$

for $n > 3$.

For this, we need to introduce some fresh variables to help us split D into multiple new conjuncts. If $n = 4$ then we may introduce x_1 to get

$$D \equiv (L_1 \vee L_2 \vee x_1) \wedge (\neg x_1 \vee L_3 \vee L_4)$$

These are equivalent, since:

- If D is true then one of the literals in D is true, which will lie in one of the two parts above, forcing that true. For the other part, we can just set the new literal involving x_1 true to make it true.
- D is not true then none of the literals are true, meaning that the conjunction of the two parts above is true if and only if x_1 and $\neg x_1$ are both true, which is not the case.

Without going into full details, we can extend this inductively so that, for $n > 4$, we have

$$D \equiv (L_1 \vee L_2 \vee x_1) \wedge \bigwedge_{i=3}^{n-2} (\neg x_{i-2} \vee L_i \vee x_{i-1}) \wedge (\neg x_{n-3} \vee L_{n-1} \vee L_n)$$

Transforming D into the above can be done in polynomial time (we claim this without proof). Hence, performing this on $O(n)$ conjuncts, we get a polynomial-computable transformation overall. Note that, for invalid CNF expressions, we can just map to Λ .

Now, since the resulting transformation Ψ is equivalent to Φ , their satisfiability is equivalent, therefore meaning we have a valid polynomial-time reduction. Hence, by 2.7, 3-Sat must be NP-hard. Combining this with it being in NP, it follows that 3-Sat is NP-complete. \square

Theorem 2.9

The G-Sat problem, stated as follows, is NP-complete:

Given a boolean expression Φ , determine whether Φ is satisfiable.

Proof. It can be shown that determining the truth value of a general boolean expression can be done in polynomial time, and therefore the exact same strategy as for Sat and 3-Sat can be applied to show that G-Sat is in NP. To show that G-Sat is NP-hard, we use the following reduction $f : \Sigma^* \rightarrow \Sigma^*$:

$$f(x) = \begin{cases} x & \text{if } x \text{ is a boolean CNF expression} \\ \Lambda & \text{otherwise} \end{cases}$$

If x is a satisfiable boolean CNF expression then so is $f(x)$. Otherwise, either $f(x) = \Lambda$ (and therefore not a yes-instance of G-Sat) or $f(x) = x$ (and therefore not satisfiable despite being in CNF). We see that f is computable in polynomial time, and therefore Sat reduces to G-Sat in polynomial time. Thus, G-Sat is also NP-hard. Combining these, we get that G-Sat is NP-complete. \square

Theorem 2.10

The clique/complete subgraph problem, stated as follows, is NP-complete:

Given a graph G and an integer $k > 0$, determine whether G has a complete subgraph consisting of k vertices.

Proof. Using guess-and-check, we know that this problem is in NP: we can nondeterministically select a particular subgraph and then check whether it is complete. As for NP-hardness, we reduce Sat to the clique problem. Suppose we have a Sat instance,

$$\Phi \triangleq \bigwedge_{i=1}^n \bigvee_{j=1}^{m_i} L_{i,j}$$

From this, we construct a set of vertices

$$V = \{L_{i,j} \mid 1 \leq i \leq n \wedge 1 \leq j \leq m_i\}$$

We then construct edges based on different conjuncts and whether two nodes are complementary:

$$E = \{\{L_{i,j}, L_{k,l}\} \mid i \neq k \text{ and } L_{i,j} \neq \neg L_{k,l}\}$$

We claim without formal proof that the algorithm for this transformation $\Phi \mapsto (V, E)$ is done in polynomial time. If Φ is satisfiable, then all of the conjuncts

$$\bigvee_{j=1}^{m_i} L_{i,j}$$

are able to be simultaneously satisfied. Hence, there must be some subset of V , say V' , given by

$$V' = \{L_{1,j_1}, L_{2,j_2}, \dots, L_{n,j_n}\}$$

where no pair of literals are complementary. Based on the definition of E , there is an edge between all pairs in V' , and therefore V' is able to form a complete subgraph from (V, E) of size n .

Conversely, suppose (V, E) has a complete subgraph (V', E') of size n . Then, by definition of E , each literal in V' must have come from a different conjunct, allowing us to write

$$V' = \{L_{1,j_1}, L_{2,j_2}, \dots, L_{n,j_n}\}$$

as before. Further, the definition of E and completeness of this subgraph tells us that all pairs of literals in V' are non-complementary. Thus, assigning these literals true is enough to guarantee all of the conjuncts true, and therefore Φ true, rendering Φ satisfiable. It follows that this mapping is a polynomial-time reduction from Sat to the clique problem, as required. \square