As for an example of a class that is not closed under polynomial-time mapping reductions, it turns out that while both P and EXP have a positive closure result, the "in-between" class E does not.

Lemma 8. *The class* E *is not closed under polynomial-time mapping reductions.*

Proof. Suppose by way of contradiction that E is closed under polynomial-time mapping reductions, and let $A \in \mathsf{EXP}$ be an arbitrary decision problem recognized by a deterministic Turing machine $\mathcal{M}$ in time $O(2^{n^c})$ for some $c \geq 1$.

Construct a deterministic Turing machine $\mathcal{M}'$ that takes a word w as input and writes to its output tape the word $w\mathsf{s}^{|w|^c}$, where s is a new tape symbol not appearing in w. Denote the language of $\mathcal{M}'$ by A_{pad}.

We can see immediately that $A \leq_m^P A_{\text{pad}}$, since we can run the computation of $\mathcal{M}$ on the prefix of the word $w\mathsf{s}^{|w|^c}$ and accept if $\mathcal{M}$ accepts. Observe also that $A_{\text{pad}} \in \mathsf{E}$, since we can recognize $x = w\mathsf{s}^{|w|^c}$ in time $O(2^{|x|})$ by first checking that x is of the correct form in time polynomial in $|x|$ and then simulating $\mathcal{M}$ on x in the manner we described earlier in time $O(2^{|x|})$.

By our assumption that E is closed under polynomial-time mapping reductions, we have that $A \in \mathsf{E}$ and so $\mathsf{EXP} \subseteq \mathsf{E}$. Since by definition we have that $\mathsf{E} \subseteq \mathsf{EXP}$, this allows us to conclude that $\mathsf{E} = \mathsf{EXP}$.

However, by the time hierarchy theorem, we know that $\mathsf{E} \subset \mathsf{EXP}$, and so we arrive at a contradiction. □

You may have noticed that our proof of Lemma 8 makes use of a padding argument. Recall that we mentioned a padding argument can also be used to prove that $\mathsf{NL} = \mathsf{coNL}$, as shown by Immerman and Szelepcsényi. Here, the purpose of adding the useless s symbols to the word $w\mathsf{s}^{|w|^c}$ was to make something already belonging to the class EXP (i.e., the w prefix of the word) "fit into" the class E as well, thus leading to our nonsensical conclusion of $\mathsf{E} = \mathsf{EXP}$.

So, some classes larger than P are not closed under polynomial-time mapping reductions. At the same time, as a consequence of Lemma 5, it is also the case that classes smaller than P are not closed under polynomial-time mapping reductions. Indeed, *any* time complexity class strictly contained within P—like L or $\mathsf{DTIME}(n)$—is not closed, and this is because the mapping reduction running in polynomial time would by its very nature blow up the resource usage to at least polynomial.

Although Lemma 8 gave us a negative result, it can lead us to good findings elsewhere. For example, since we know that PSPACE is closed under polynomial-time mapping reductions but E is not, we obtain a nice separation in our fundamental complexity hierarchy:

Corollary 9. $\mathsf{E} \neq \mathsf{PSPACE}$.

Naturally, in addition to $\mathsf{E} \subset \mathsf{EXP}$, we know that $\mathsf{P} \subset \mathsf{E}$ by the time hierarchy theorem. However, to build some mystery into our complexity hierarchy, nobody yet knows whether $\mathsf{E} \subset \mathsf{PSPACE}$ or $\mathsf{PSPACE} \subset \mathsf{E}$, or whether the two classes are even comparable!

$$\begin{array}{ccccc} & \subset & \mathsf{E} & \subset & \\ & & \uparrow & & \\ \mathsf{P} & & ? & & \mathsf{EXP} \\ & & \downarrow & & \\ & \subseteq & \mathsf{PSPACE} & \subseteq & \end{array}$$

2 Hardness and Completeness

As we observed, reductions are not symmetric; being able to reduce from one decision problem to another does not necessarily imply that we can reduce the other way around, which would show that the two decision problems are equivalent. However, if we consider a particular complexity class, say NP, then we can use reductions to establish the *hardness* of decision problems relative to that class without necessarily needing to show the equivalence of any two decision problems.

We can establish some measure of hardness as follows: if every decision problem in NP can be reduced to one particular decision problem A, then we can count A as having a difficulty on par with any other decision problem in NP. Furthermore, the decision problem A doesn't necessarily have to belong to NP itself in order to be as hard as any decision problem in NP; all we are saying is that any decision problem in NP can be modelled as an instance of A, so if we could solve A, then we could also solve any decision problem in NP.

Speaking more generally, if we're able to reduce any decision problem in some complexity class C to a particular decision problem A, then A is at least as hard as any other decision problem in C, and so we say as a shorthand that A is a *C-hard* decision problem.

Definition 10 (C-hardness)**.** For any complexity class C, a decision problem A is said to be C-hard if, for every decision problem $B \in \mathsf{C}$, there exists a mapping reduction $B \leq_m A$.

In the same spirit as the notion of hardness, the notion of a decision problem being *complete* for a complexity class indicates that such a decision problem is representative of its entire complexity class with respect to difficulty. While we noted that a C-hard decision problem A doesn't necessarily need to belong to the class C itself, if we additionally know that A is in the class C, then we say that A is a *C-complete* decision problem.

Definition 11 (C-completeness)**.** For any complexity class C, a decision problem A is said to be C-complete if $A \in \mathsf{C}$ and A is C-hard.

Since we know by Lemma 3 that mapping reductions are transitive, the notions of hardness and completeness allow us to order decision problems according to their difficulty. In particular, any decision problem that is C-complete is a maximal element of C with respect to this difficulty ordering; in other terms, C-complete problems are the "toughest of the tough" problems in the entire class C.[1]

Note that, in both Definitions 10 and 11, we did not indicate explicitly that the mapping reductions should take polynomial time. Rather, we can determine via context what sort of mapping reduction is most appropriate; for instance, if we are talking about some decision problem being hard or complete for a complexity class like NP, we should assume that we are using a polynomial-time mapping reduction to establish hardness or completeness.

On the other hand, as a consequence of Lemma 5, the notion of completeness with respect to polynomial-time mapping reductions does not give us anything useful when we're discussing smaller complexity classes like P. Although we can still reason about notions like P-completeness, we will need to use a different kind of mapping reduction to do so.

2.1 NP-Completeness

Let's take a closer look at completeness through the lens of the complexity class NP. The notion of NP-completeness is among the most important in all of complexity theory, since it not only encapsulates what we intuit to be the "most difficult" problems for computers to solve in a reasonable amount of time, but it could also hold the key to answering some of the field's biggest questions. For instance, since every decision problem in NP reduces to an NP-complete decision problem, if we were able to show that *just one* NP-complete decision problem could be solved in polynomial time, then we would prove that $\mathsf{P} = \mathsf{NP}$. Since we don't yet have an answer to this long-standing question, knowing that a decision problem is NP-complete serves as strong evidence that the decision problem is not in P.

However, from the wording of Definition 11 itself, it is not immediately obvious that NP-complete decision problems even exist. In fact, the notion of NP-completeness was developed and studied for years before anyone was able to produce an example of an NP-complete decision problem.

In 1971, the American-Canadian computer scientist Stephen Cook and the Soviet mathematician Leonid Levin independently published the same remarkable result: the *Boolean satisfiability problem* is NP-complete.

[1] Note that a decision problem being C-complete only suggests that it is among the toughest problems to solve *within the class C*, not that it is among the toughest problems in general. For example, NP-complete decision problems take at most exponential time to solve on a deterministic Turing machine, but many other decision problems provably require more than exponential time to obtain a solution. Some decision problems, like the halting problem, can't even be solved in all instances!

The Boolean satisfiability problem, or SATISFIABILITY for short, asks whether there exists some assignment of true and false values to Boolean variables that satisfies every clause in a given Boolean formula.

Before we proceed further, let's clarify some terminology. A *Boolean formula* is a logical combination of *Boolean variables*. Variables are arranged in *clauses*; for example, the formula $(x_1 \vee x_2) \wedge (x_3 \vee x_4)$ contains four Boolean variables and two clauses. This example formula is also said to be in *conjunctive normal form*, since each clause contains only $\vee$s, and clauses are joined using only $\wedge$s. Lastly, a *satisfying assignment* of a Boolean formula is one that renders the overall formula true; for example, in our previous formula, $x_1 = x_3 =$ true and $x_2 = x_4 =$ false is a satisfying assignment.

The formal statement of the Boolean satisfiability problem, then, is as follows.

SATISFIABILITY
Given: a Boolean formula $C_1 \wedge C_2 \wedge \cdots \wedge C_n$ in conjunctive normal form
Determine: whether there exists an assignment of truth values to Boolean variables satisfying all clauses in the Boolean formula

How did Cook and Levin show that the Boolean satisfiability problem is NP-complete? Well, showing that the problem is in NP is the easy step. The proof of NP-hardness, though, is quite clever: using the fact that the class NP contains all decision problems that can be decided in polynomial time by a nondeterministic Turing machine, one simply constructs a Boolean formula that simulates the computation of such a nondeterministic Turing machine on a given input word. If this simulated machine accepts, then the Boolean formula has a satisfying assignment!

The complete proof of NP-hardness is quite long and technical, and since we don't really require it for anything else, we'll only present a sketch of that part of the proof. The proof of membership in the class NP, however, only takes a few lines.

Theorem 12 (Cook–Levin theorem)**.** SATISFIABILITY *is* NP*-complete.*

Proof Sketch. We begin by showing that SATISFIABILITY is in NP. This step is straightforward: we can construct a polynomial-time nondeterministic Turing machine $\mathcal{M}_{\text{SAT}}$ that takes as input a Boolean formula ϕ and guesses a satisfying assignment of values to variables. If the assignment is in fact satisfying, then $\mathcal{M}_{\text{SAT}}$ accepts.

Next, we sketch the proof that SATISFIABILITY is NP-hard. To do this, consider any decision problem $L \in \mathsf{NP}$. We know that a polynomial-time nondeterministic Turing machine $\mathcal{M}_L$ exists that decides L in time n^k for some $k \geq 0$.

If $n = |w|$, where w is the input word given to $\mathcal{M}_L$, then any computation of $\mathcal{M}_L$ on w has at most n^k configurations. Suppose we take all such configurations and create a computation table of size $n^k \times n^k$. Each index of the computation table contains a symbol from the set $C = Q \cup \Gamma \cup \{\#\}$, where $\#$ is a special boundary marker written on the left and right sides of the computation table.

We can represent the contents of each index (i, j) of the computation table by $|C|$ Boolean variables, each of the form $\{x_{i,j,s} \mid s \in C\}$. If $x_{i,j,s} =$ true, then this variable indicates the symbol at index (i, j) of the computation table is s. In total, we require $|C| \cdot n^{2k}$ Boolean variables.

Next, using our set of Boolean variables, we create Boolean formulas to "verify" the computation of $\mathcal{M}_L$. We require our formulas to satisfy four conditions:

$$\begin{aligned}
\phi_{\text{start}} &= \{\text{the first row of the computation table is the start configuration of } \mathcal{M}_L \text{ on } w\}; \\
\phi_{\text{acc}} &= \{\text{the last row of the computation table is an accepting configuration of } \mathcal{M}_L \text{ on } w\}; \\
\phi_{\text{idx}} &= \{\text{for all indices } (i,j)\text{, there exists exactly one } s \in C \text{ such that } x_{i,j,s} = \text{true}\}; \text{and} \\
\phi_{\text{tran}} &= \{\text{each } 2 \times 3 \text{ subblock of the computation table satisfies the transition function of } \mathcal{M}_L\}.
\end{aligned}$$

For each of these four conditions, we can create a Boolean formula of size $O(n^{2k})$ to express the condition, and we can construct the formula in polynomial time relative to the input word w.

Finally, we claim that $\mathcal{M}_L$ has an accepting configuration on w if and only if the Boolean formula $\phi_{\mathcal{M}_L} = \phi_{\text{start}} \wedge \phi_{\text{acc}} \wedge \phi_{\text{idx}} \wedge \phi_{\text{tran}}$ has a satisfying assignment. In this way, we have developed a polynomial-time reduction $L \leq_m^P$ Satisfiability. Since we can do this for every decision problem $L \in \mathsf{NP}$, we conclude that Satisfiability is NP-hard. □

Fortunately, showing that other decision problems are NP-complete is not as involved a process as it is for Satisfiability. Thanks to the transitivity of polynomial-time mapping reductions, together with the fact that NP is closed under such reductions, we have a rather easy way of establishing the NP-completeness of other decision problems in NP: we just need one mapping reduction from a known NP-complete decision problem to our new decision problem.

Theorem 13. *Let A and B be decision problems. If $A \leq_m^P B$, A is NP-complete, and $B \in \mathsf{NP}$, then B is NP-complete.*

Proof. Follows immediately from Lemmas 3 and 7. □

As a result, since we know that Satisfiability is NP-complete, we can build a "tree of reductions" rooted at Satisfiability to show that hundreds of other decision problems belonging to NP are additionally NP-complete.

2.2 PSPACE-Completeness

What of completeness for other complexity classes? If we shift our focus from (nondeterministic) polynomial time to polynomial space, then we can naturally study the notion of PSPACE-completeness in much the same way as we studied NP-completeness. The class of PSPACE-complete decision problems is generally suspected to lie outside of either of the classes P or NP, though nobody has been able to prove this yet; indeed, if we were again able to show that *just one* PSPACE-complete decision problem belonged to either P or NP, then we would prove that $\mathsf{P} = \mathsf{PSPACE}$ or $\mathsf{NP} = \mathsf{PSPACE}$, respectively, and the importance of this question is on par with that of the P vs. NP question.

Just as we did for establishing NP-completeness, we will rely on polynomial-time mapping reductions to establish PSPACE-completeness. Of course, one might wonder why we're using polynomial-*time* mapping reductions instead of polynomial-*space* mapping reductions, given that we're now talking about the class of polynomial *space* decision problems. The simple answer is that we want to be able to compute our reductions efficiently; that is, in polynomial time. If all we know is that our reductions require a polynomial amount of space, then it may be the case that we need exponential time (or worse) just to perform the reduction—in turn, this would mean that an efficient method of solving a PSPACE-complete decision problem might not be so efficient after performing the reduction to that decision problem! Using polynomial-time mapping reductions ensures that the process of reducing is no more difficult than the process of solving.

As with NP-completeness, though, Definition 11 by itself does not suggest the existence of a PSPACE-complete decision problem. In 1964, the Japanese linguist Sige-Yuki Kuroda showed that the *membership problem* for deterministic context-sensitive grammars was PSPACE-complete. In a nutshell, this problem asks whether some word w can be produced through a finite sequence of applications of rules from a given deterministic context-sensitive grammar; in other terms, does w belong to the language of the grammar?

The fact that testing membership in a deterministic context-sensitive grammar is PSPACE-complete is somewhat surprising, since the class of deterministic context-sensitive languages is equal to the complexity class $\mathsf{DSPACE}(n)$, and while linear space is still technically polynomial, it's a very small polynomial. We are able to make this problem "fill out" the rest of PSPACE by using a standard padding argument as we did before.

Here, however, we will focus our attention on another decision problem that is often used as the canonical example of a PSPACE-complete decision problem. Recall from before our discussion of Boolean formulas and variables. We can modify a Boolean formula by applying *quantifiers* to variables in the formula. Given a variable x, the *existential quantifier* $\exists x$ says that there exists *some* assignment to x that renders the formula true, while the *universal quantifier* $\forall x$ says that *all* assignments to x render the formula true.

Within a formula, we may quantify some or all of its variables; for example, the formula $\exists x_2\ (x_1 \wedge x_2) \vee \forall x_4\ (x_3 \vee x_4)$ and the formula $\forall x_1\ \exists x_2\ (x_1 \wedge x_2)$ are both quantified formulas. If all of the quantifiers are in front of the formula, we say the formula is in *prenex normal form*, and it is straightforward to convert an arbitrary quantified Boolean formula to one in prenex normal form.

If, additionally, it is the case that all of the variables in a formula are quantified, then we say that it is a *fully quantified Boolean formula*. Such formulas are always either true or false, and this gives rise to the decision problem we consider here. The *true fully quantified Boolean formula problem*, or TQBF for short, asks whether a given fully quantified Boolean formula is true.

TQBF
Given: a fully quantified Boolean formula $Q_1 x_1\ Q_2 x_2\ \ldots\ Q_n x_n\ \phi(x_1, x_2, \ldots, x_n)$, where $Q_i \in \{\exists, \forall\}$ for all $1 \leq i \leq n$ Determine: whether ϕ is true

Example 14. The following fully quantified Boolean formula is true:

$$\phi = \forall x_1\ \exists x_2\ ((x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2)).$$

Suppose that $x_1 =$ true. Then ϕ is satisfied by the assignment $x_2 =$ false, since the first clause $(x_1 \vee x_2)$ is satisfied by x_1 and the second clause $(\neg x_1 \vee \neg x_2)$ is satisfied by x_2. Otherwise, if $x_1 =$ false, then ϕ is likewise satisfied by the assignment $x_2 =$ true. In other words, for all assignments to x_1, there exists an assignment to x_2 that satisfies ϕ.

Example 15. The following fully quantified Boolean formula is false:

$$\phi = \forall x_1\ \exists x_2\ \forall x_3\ ((x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_3)).$$

Suppose that $x_1 =$ true; it can be either true or false, but due to the presence of the quantifier $\forall x_1$, either assignment must satisfy ϕ. In order for the second clause $(\neg x_1 \vee x_3)$ to be true, it must be the case that $x_3 =$ true. However, since ϕ also contains the quantifier $\forall x_3$, we cannot fix an assignment for x_3, and it turns out that if both $x_1 =$ true and $x_3 =$ false, then ϕ is not satisfied.

Amazingly, despite the simple formulation of this decision problem, it is in fact PSPACE-complete! While it is easy to show that the decision problem belongs to the class PSPACE, we will again only sketch the proof of PSPACE-hardness.

Theorem 16. TQBF *is* PSPACE*-complete.*

Proof Sketch. We begin by showing that TQBF is in PSPACE. To do this, we consider three cases depending on the Boolean formula ϕ we receive as input:

- If ϕ has no quantifiers, then evaluate ϕ directly. If ϕ is true, then accept. Otherwise, reject.
- If $\phi = \exists x_i\ \psi$ where ψ is some subformula, then evaluate ψ twice: once with $x_i =$ true, and once with $x_i =$ false. If either evaluation is true, then accept. Otherwise, reject.
- If $\phi = \forall x_i\ \psi$ where ψ is some subformula, then evaluate ψ twice: once with $x_i =$ true, and once with $x_i =$ false. If both evaluations are true, then accept. Otherwise, reject.

Since this procedure recurses at most m times, it uses $O(m)$ space, where m is the number of Boolean variables in ϕ. Thus, the procedure runs in polynomial space.

Next, we sketch the proof that TQBF is PSPACE-hard. To do this, consider any decision problem $L \in \mathsf{PSPACE}$. We know that a polynomial-space Turing machine $\mathcal{M}_L$ exists that decides L in space n^k for some $k \geq 0$.

We will encode the computation of $\mathcal{M}_L$ on an input word w of length n as a fully quantified Boolean formula. Since we know $\mathcal{M}_L$ decides L in space n^k, individual configurations of $\mathcal{M}_L$ can be encoded using $O(n^k)$ bits.

More precisely, given the input word w, we construct a sequence of fully quantified Boolean formulas $\phi_k(C_a, C_b)$, where any given formula $\phi_k(C_a, C_b)$ is true if and only if configuration C_b is reachable from configuration C_a in at most 2^k steps. Then, the formula $\phi_{n^k}(C_0, C_{\text{acc}})$ is true if and only if $\mathcal{M}_L$ accepts its input word w. (We use ϕ_{n^k} since there is a total of 2^{n^k} configurations for a computation using space n^k.)

The only question remaining is how we construct each fully quantified Boolean formula. Taking ϕ_0 as our base case, we can evaluate $\phi_0(C_a, C_b)$ by checking (i) whether $C_a = C_b$; or (ii) whether the edge (C_a, C_b) exists in the computation graph of $\mathcal{M}_L$. For the inductive case, to construct the formula ϕ_{i+1} from the formula ϕ_i, the most straightforward method would be to take

$$\phi_{i+1}(C_a, C_b) = \exists C_x \; \phi_i(C_a, C_x) \wedge \phi_i(C_x, C_b). \tag{1}$$

However, following this construction would result in the formula ϕ_{n^k} having exponential length, since this process of constructing ϕ_{i+1} from two instances of ϕ_i doubles the length of the formula.

Instead, by using a particular encoding of the necessary information, we can construct the formula ϕ_{i+1} using just one instance of ϕ_i and not two:

$$\phi_{i+1}(C_a, C_b) = \exists C_x \; \forall x, y \; \Big(\big((x, y) = (C_a, C_x)\big) \vee \big((x, y) = (C_x, C_b)\big)\Big) \Rightarrow \phi_i(x, y) \tag{2}$$

In Formula 2, we express all the same information as in Formula 1, but in a more concise manner. We assert that there exist two pairs (x, y) such that $\phi_i(x, y)$ is true, and these two pairs are exactly (C_a, C_x) and (C_x, C_b) for some intermediate configuration C_x. Although quantifiers may appear within ϕ_i, it is straightforward for us to shift those quantifiers to the front of ϕ_{i+1} to render it in prenex normal form.

Observe that, while representing ϕ_{n^k} in the style of Formula 1 would result in it having exponential length, representing ϕ_{n^k} in the style of Formula 2 keeps the length of the formula bounded by a polynomial. This is because each formula ϕ_{i+1} is only an additive factor of $O(n^k)$ longer than its predecessor, ϕ_i. Thus, the formula ϕ_{n^k} can be computed using polynomial space. $\square$

As we would expect, if we know of one PSPACE-complete decision problem, then it is easy for us to prove that other decision problems within PSPACE are also PSPACE-complete.

Theorem 17. *Let A and B be decision problems. If $A \leq_m^P B$, A is* PSPACE*-complete, and $B \in$* PSPACE*, then B is* PSPACE*-complete.*

Proof. Follows immediately from Lemmas 3 and 7. $\square$

Examples of PSPACE-complete problems abound in the literature, and interestingly, not all such problems pertain directly to computers—many PSPACE-complete problems belong to the realm of board games, video games, and puzzles.

3 Logarithmic-Space Mapping Reductions

In the previous sections, we repeatedly mentioned the fact that polynomial-time mapping reductions are inappropriate to use if we want to prove properties about the class P or its subclasses, like L and NL. This is because, for these classes, polynomial-time mapping reductions are too coarse: we already know by Lemma 5 that we can use polynomial-time mapping reductions to reduce a decision problem $A \in$ P to any non-trivial language, and we can similarly solve any decision problem belonging to either L or NL in polynomial time.

In general, we avoid using mapping reductions that are as powerful as (or more powerful than) the complexity class we're talking about, since they don't tell us anything new or interesting about that class. Using a mapping reduction that is weaker than the complexity class we're talking about allows us to establish worthwhile results about that class, since applying the reduction requires a lesser amount of work than solving the decision problem itself. Thus, polynomial-time mapping reductions are appropriate for classes like NP and PSPACE, but not for P, L, or NL.

So, what can we use for these smaller complexity classes? We can still use mapping reductions, but not reductions that run in polynomial time. Instead, we will define a new mapping reduction that uses a function computable by a Turing machine using a small amount of space.

Definition 18 (Logarithmic-space mapping reduction)**.** Given two decision problems A and B, problem A is logarithmic-space mapping reducible to problem B if there exists a logarithmic-space computable function $f\colon \Sigma^* \to \Sigma^*$ where, for all $w \in \Sigma^*$, $w \in A$ if and only if $f(w) \in B$.

As we had with polynomial-time mapping reductions, if A is logarithmic-space reducible to B, then we can transform every instance w of A to an instance $f(w)$ of B using logarithmic space. We denote a logarithmic-space mapping reduction from A to B by the notation $A \leq_m^L B$.

Recall that, since we're considering space complexity here, we only measure the space usage of our mapping reduction in terms of the number of cells used on the work tapes of our Turing machine. We assume like before that the input tape is read-only and does not contribute to the space usage of the computation. Additionally, we will assume that the result of the mapping reduction is written to a special write-only and write-once output tape, which also does not contribute to the space usage. A Turing machine with this additional condition applied to the output tape is known as a *logarithmic-space transducer*, and such machines are specially designed to compute logarithmic-space mapping reductions.

3.1 Properties

Just as we did with polynomial-time mapping reductions, we can prove some properties about logarithmic-space mapping reductions. The first property is more-or-less immediate.

Lemma 19. *The logarithmic-space mapping reduction relation $\leq_m^L$ is reflexive.*

Proof. Take $f(x) = x$ as our computable function; since neither the input nor output tapes contribute to the space usage, writing the input word directly to the output tape uses a constant (and therefore logarithmic) amount of space on the work tapes. Thus, $A \leq_m^L A$ for all decision problems A. □

As before, the mapping reduction relation is *not* symmetric.

We can still prove transitivity, but the proof is a bit more involved than that for polynomial-time mapping reductions. Before, we could simply "feed in" the result of the first reduction directly to the second reduction, since both reductions operated in polynomial time. However, since we're now restricted to using only logarithmic space, we can't even store the result of the first reduction in order to give it to the second reduction, as the mere act of writing down the result requires a polynomial amount of space! We must instead use a more clever approach.

Lemma 20. *The logarithmic-space mapping reduction relation $\leq_m^L$ is transitive.*

Proof. Suppose that $A \leq_m^L B$ by way of some logarithmic-space Turing machine $\mathcal{M}_1$ computing a function f, and $B \leq_m^L C$ by way of some logarithmic-space Turing machine $\mathcal{M}_2$ computing a function g. We can define a Turing machine $\mathcal{M}_3$ that takes as input a word w and computes $g(f(w))$, but we run into a problem: $\mathcal{M}_3$ cannot write down the intermediate result $f(w)$, as the length of this result is polynomial in $|w|$.

Instead, we will define $\mathcal{M}_3$ to work in the following way:

1. On one work tape, simulate the computation of $\mathcal{M}_1$ on the input word w, but *do not* write down the output of $\mathcal{M}_1$.
2. On two other work tapes, simulate the computation of $\mathcal{M}_2$ on one work tape given access to the position of the tape head of $\mathcal{M}_1$ stored as a binary counter on another work tape.
3. After each computation step, adjust the positions of the input heads and restart the computation of $\mathcal{M}_1$ from the beginning.

The first step clearly uses logarithmic space, because $\mathcal{M}_1$ is a logarithmic-space Turing machine. The second step also uses logarithmic space, both because $\mathcal{M}_2$ is also a logarithmic-space Turing machine and because we record the position of the tape head of $\mathcal{M}_1$ using a binary counter. Thus, the overall computation of $\mathcal{M}_3$ uses logarithmic space. □

In essence, the proof of Lemma 20 has us computing individual bits of $f(w)$ "on-the-fly" in order to compute $g(f(w))$ without exceeding the logarithmic space limit. Since we can't store all of $f(w)$ on a work tape, we simulate having $f(w)$ on a virtual tape and, whenever we need to read a bit from that virtual tape, we compute that bit directly. While this ensures we fit into the space requirement, we unfortunately must trade time for space; if the tape head of $\mathcal{M}_1$ ever moves left, for example, we can't remember that symbol and so we must re-run the entire computation of $\mathcal{M}_1$ to read it.

Lastly, it is possible for us to relate our two mapping reductions to each other in that if we have a logarithmic-space mapping reduction between two decision problems, then we necessarily also have a polynomial-time mapping reduction between the same decision problems.

Lemma 21. *Given two decision problems A and B, if $A \leq_m^L B$, then $A \leq_m^P B$.*

Proof. Since $\mathsf{DSPACE}(f(n)) \subseteq \mathsf{NSPACE}(f(n)) \subseteq \mathsf{DTIME}\big(2^{O(f(n))}\big)$, any Turing machine that uses a logarithmic amount of space also uses a polynomial amount of time. Thus, any logarithmic-space mapping reduction is also a polynomial-time mapping reduction. □

Since we don't know whether $\mathsf{L} = \mathsf{P}$, it remains unknown whether the converse of Lemma 21 holds, which would tell us that every polynomial-time mapping reduction has a corresponding logarithmic-space mapping reduction.

In practice, we can often substitute logarithmic-space mapping reductions for polynomial-time mapping reductions. For example, we can prove the Cook–Levin theorem using only logarithmic-space mapping reductions. In fact, there are no known decision problems that are NP-complete with respect to polynomial-time mapping reductions that are not also NP-complete with respect to logarithmic-space mapping reductions; if there were, then this would suggest that $\mathsf{L} \neq \mathsf{P}$.

Note, however, that even though we can prove the Cook–Levin theorem using logarithmic-space mapping reductions, this does not by itself imply that SATISFIABILITY is in either L or NL. Indeed, if we could prove either of these results, then we would establish that either $\mathsf{L} = \mathsf{NP}$ or $\mathsf{NL} = \mathsf{NP}$; the former would be a remarkable result, given that nobody to date has been able to establish whether nondeterministic polynomial-time computations are more powerful than deterministic logarithmic-space computations!

3.2 Closure

Finally, we prove the all-important property of closure for logarithmic-space mapping reductions. When we considered polynomial-time mapping reductions, we saw that each of the complexity classes P, NP, PSPACE, and EXP were closed under that relation, but smaller classes like L were not closed. With logarithmic-space mapping reductions, on the other hand, we can establish closure for logarithmic space complexity classes.

Lemma 22. *The classes L and NL are closed under logarithmic-space mapping reductions.*

Proof. We will prove closure for the class L; the proof for NL is similar.

Let A and B be decision problems. Suppose that $A \leq_m^L B$ by way of some logarithmic-space deterministic Turing machine $\mathcal{M}$ computing a function f. Further suppose that $B \in \mathsf{L}$; that is, suppose some logarithmic-space deterministic Turing machine $\mathcal{N}$ recognizes B.

We use the same construction as in the proof of Lemma 20, where we take the first reduction to be the function f computed by $\mathcal{M}$ and we take the second reduction to be the characteristic function (say, g) of B; that is, $g(w) = 1$ if and only if $w \in B$.

Specifically, we construct a deterministic Turing machine $\mathcal{N}'$ that, given an input word w, simulates the computation of $\mathcal{N}$ on the input word $f(w)$ by computing individual bits of $f(w)$ "on-the-fly". Then, $\mathcal{N}'$ accepts w if and only if $g(f(w)) = 1$; that is, $w \in A$ if and only if $f(w) \in B$.

Since both the mapping reduction and the computation of the characteristic function use at most a logarithmic amount of space, the overall computation of $\mathcal{N}'$ uses logarithmic space, and so $A \in \mathsf{L}$. □

It is not too difficult to prove that each of P, NP, PSPACE, and EXP are closed under logarithmic-space mapping reductions as well. However, some small classes like $\mathsf{DTIME}(n)$ are still not closed even for logarithmic-space mapping reductions.