CS331: Algorithms and Complexity
Part VIII: Complexity Theory

Kevin Tian

1 Introduction

In this course, we have discussed polynomial-time algorithms for a broad range of tasks. This includes basic primitives like sorting, reachability, and matrix multiplication, as well as much more challenging problems such as all-pairs shortest paths, maxflow, and linear programming. Recall that we call an algorithm *polynomial-time* if on an input of length n, it solves a given computational problem in time $f(n)$, where $f(n)$ is a (constant-degree) polynomial in n.

The flipside of this story is that there are many fundamental problems for which, despite humanity's best efforts, there remains no known polynomial-time algorithm. We give a few examples with particular relevance to problems already discussed in this course.

- **Subset sum.** Given an Array of positive integers $A = \{a_i \in \mathbb{N}\}_{i\in[m]}$ and a target value $T \in \mathbb{N}$, determine whether there is a subset $S \subseteq [m]$ such that $\sum_{i\in S} a_i = T$. We gave a *dynamic programming*-based algorithm for this problem in Section 3.3, Part III that runs in time $O(mT)$. Unfortunately, if we let n be the length (in bits) of the input (A, T), the actual value of T could be as large as $\exp(\Omega(n))$, so the DP-based algorithm is not polynomial-time in general. We have yet to discover a polynomial-time algorithm for subset sum.
- **Maximum cut.** Given a graph $G = (V, E)$, find a subset $S \subseteq V$ such that $\text{cut}(S)$ is as large as possible. Recall from Eq. (9), Part V that $\text{cut}(S)$ is the number of edges in E crossing from S to its complement, $V \setminus S$. We learned in Section 4, Part V that the *minimum cut* problem is solvable in polynomial time; no such algorithm is known for *maximum cut*.
- **Vertex cover.** Given a graph $G = (V, E)$, find as small of a subset $S \subseteq V$ as possible such that every edge $e \in E$ has at least one endpoint in S. This problem can be rewritten as: find $\mathbf{x} \in \{0, 1\}^V$ such that $\sum_{v\in V} \mathbf{x}_v$ is minimal, and $\mathbf{x}_u + \mathbf{x}_v \geq 1$ for all $e = (u, v) \in E$. This equivalence is because we can choose $\mathbf{x}$ to set all vertices in S to 1, at which point $\mathbf{x}_u + \mathbf{x}_v \geq 1$ implies one endpoint of every edge is in S. If we change the constraint $\mathbf{x} \in \{0, 1\}^V$ to $\mathbf{x} \in [0, 1]^V$, i.e., we allow for fractional values, then this problem is a *linear program* and hence can be solved in polynomial time using algorithms from Section 5.3, Part VI. No such algorithm is known for the variant where $\mathbf{x}$ must have integer values.

What makes some problems (e.g., those discussed in Parts I to VII of the course) easy, and other problems (e.g., those discussed above) hard? This is exactly one of the main questions asked by *complexity theory*. The goal of (computational) complexity theory is to explore fundamental distinctions in the complexity of different problems in computer science. These notes provide a basic introduction to this deep subject with wide-ranging implications.

A common theme in complexity theory, that we hope to expose in these notes, is the notion of a *barrier*. A barrier is, informally, a common source of hardness for an entire class of problems, algorithms, or techniques. For example, the most famous unsolved problem in theoretical computer science (the subject of Sections 2 and 3) is the P vs. NP problem, which asks whether two complexity classes, P and NP, are the same. While we do not know the answer to this question, a major success story of complexity theory is the notion of *NP-completeness*, the subject of Section 4. We can definitively say that designing a polynomial-time algorithm for any NP-complete problem is a barrier to establishing P = NP (see Corollary 1 for a formal statement). Barriers are an incredibly useful way to make progress on separating complexity classes and establishing a taxonomy of algorithmic problems, one of the ultimate goals of complexity theory.

2 Reductions

Before we formalize the notion of an algorithmic problem being a barrier for a complexity class, we first need to introduce *polynomial-time (polytime) reductions*.

Consider two algorithmic problems, A and B, and suppose AlgoForB is an algorithm that solves B. We say that there is a polytime reduction from A to B if there is an algorithm that solves A on length-n puts by using:

- poly(n)[1] calls to AlgoForB, and
- poly(n) extra time.

We use the helpful notation $\mathsf{A} \leq_{\mathrm{P}} \mathsf{B}$ as shorthand for "there is a polytime reduction from A to B."

For intuition, it is helpful to imagine that we have a magical box AlgoForB, such that if we feed an input for B into this magical box, it will tell us the answer for B on this input. We call this magical box an *oracle* for B. A polytime reduction from A to B is just a regular polytime algorithm for A, that is also allowed to query the oracle for B a polynomial number of times.

In fact, we have already seen several polytime reductions in Section 5.3, Part V, when we discussed *maxflow reductions*. Specifically, we established polytime reductions from the DisjointPaths and BipartiteMatching problems to the Maxflow problem. Our reduction from DisjointPaths to Maxflow, for example, took an input to DisjointPaths, and repackaged it into a new input to Maxflow. We then called Maxflow on this new input, and from the result we deduced a solution to our original DisjointPaths problem. This is exactly a polytime reduction showing that $\mathsf{DisjointPaths} \leq_{\mathrm{P}} \mathsf{Maxflow}$; we similarly demonstrated that $\mathsf{BipartiteMatching} \leq_{\mathrm{P}} \mathsf{Maxflow}$.

Polytime reductions are useful exactly because they capture the notion of a *bottleneck problem* for establishing that a class of problems admit polytime algorithms.

Lemma 1. *Suppose that* $\mathsf{A} \leq_{\mathrm{P}} \mathsf{B}$*, and there is a polynomial-time algorithm for solving* B*. Then there is a polynomial-time algorithm for solving* A*.*

Proof. Let AlgoForB be the polytime algorithm for solving B, and let its runtime be $f(n) = \text{poly}(n)$. Because $\mathsf{A} \leq_{\mathrm{P}} \mathsf{B}$, we know there is an algorithm for solving A on length-n inputs that uses $g(n) = \text{poly}(n)$ time, and makes $h(n) = \text{poly}(n)$ calls to AlgoForB. The total runtime used by this algorithm is then $f(n) \cdot h(n) + g(n)$, which is still a polynomial in n. □

To interpret this statement, suppose we have established $\mathsf{A}_1, \mathsf{A}_2, \mathsf{A}_3, \ldots \leq_{\mathrm{P}} \mathsf{B}$. Then by Lemma 1, to design polytime algorithms for all of $\mathsf{A}_1, \mathsf{A}_2, \mathsf{A}_3, \ldots$, it is enough to design a single polytime algorithm for B! Thus, solving B in polytime is formally a bottleneck towards polytime algorithms for an entire class of problems. As a consequence, we should focus our attention on designing faster algorithms for B. We implicitly used this logic in Section 5.3, Part V, by showing that $\mathsf{DisjointPaths}, \mathsf{BipartiteMatching} \leq_{\mathrm{P}} \mathsf{Maxflow}$, and then claiming there exist polytime algorithms for the former two problems, because there is a polytime algorithm for Maxflow.

Decision problems. We pause briefly to define the scope of algorithmic problems we consider. These notes (and a large portion of complexity theory) focuses specifically on binary *decision problems*. These are problems L that take as input a binary string $x \in \{0,1\}^n$, and have a yes-no answer. If x is a "yes" instance, then we write $x \in \mathsf{L}$, and if it is a "no" instance, we write $x \notin \mathsf{L}$. We say that an algorithm *solves*, or *decides*, the problem L if it takes as input some $x \in \{0,1\}^n$, and determines whether x is a "yes" instance or a "no" instance.

In the rest of these notes, we use $\{0,1\}^* = \bigcup_{n \in \mathbb{N}} \{0,1\}^n$ to denote the set of binary strings with arbitrary length. For any $x \in \{0,1\}^*$, we use $|x|$ to denote its length.

Focusing on decision problems may seem restrictive, due to their binary outcomes. While decision problems do capture some problems we have studied, e.g., reachability or subset sum, many problems ask to return a numerical value (i.e., they are optimization problems rather than decision problems). This includes Maxflow, as well as other problems such as Knapsack (Section 3.3, Part III) or LongestCommonSubsequence (Section 4.1, Part III).

[1] In these notes, we will use poly(n) to denote any function $f(n)$ that is a polynomial in n. For example, n^2, $10n^3$, and more generally any $O(n^k)$ for a constant k are all examples of functions that are poly(n).

Luckily, there is usually a simple way to reduce any optimization problem to a decision problem, by passing in a threshold value as an auxiliary input. For example, determining the s-t maxflow value on a graph $G = (V, E)$ with capacities $\mathbf{c} \in \mathbb{R}^E_{\geq 0}$ is an optimization problem. However, we could instead ask: for a given threshold $T \in \mathbb{R}_{\geq 0}$, is the maxflow value in the instance $(G, \mathbf{c}, s, t)$ at least T? If we treat the input as $(G, \mathbf{c}, s, t, T)$, this new problem has a "yes" or "no" answer, so it is a decision problem. If we have a fast algorithm for the decision variant of the problem, binary searching over T lets us efficiently solve the optimization variant as well. So, without loss of much generality, we henceforth restrict our attention in these notes to decision problems.

Finally, we briefly remark on the requirement that inputs must be binary strings $x \in \{0,1\}^n$. For any natural computer science problem, there are canonical ways to encode its input in binary (indeed, at the hardware level, all inputs to algorithms ultimately are represented in bits). For example, if $G = (V, E)$ is a graph, a canonical encoding could be to write down the sizes of $|V|$ and $|E|$ in binary, and then concatenate the two endpoints of each edge $e = (u, v) \in E$ (as binary numbers indexing vertices u, v). In all standard settings, this encoding does not blow up the input size by more than a polynomial factor. Thus, whether we think of the input length n as the "size of G" or the "number of bits in the encoding x of G," the notion of a polytime algorithm does not change. We devote no further time to the specific encoding of "human-readable inputs" (e.g., the graph G) into a "decision problem input" (e.g., the binary string x representing G in bits), and simply fix some conventional encoding for each problem. We write $x = \langle G \rangle$ to signify that x is an encoding of the object G that the algorithm is meant to work with.

The reader can imagine that as a first step, any algorithm for a decision problem first decodes its input into a human-friendly format. For example, the decision variant of the s-t maxflow problem $\mathsf{Maxflow}$ takes as input a binary encoding $x = \langle G, \mathbf{c}, s, t, T \rangle$, where $\mathbf{c}$ is a vector of edge capacities, s, t are vertices, and T is a target value. An algorithm that decides $\mathsf{Maxflow}$ first checks whether its input x is actually an encoding of the form $\langle G, \mathbf{c}, s, t, T \rangle$. If it is not, the algorithm immediately declares that x is a "no" instance; otherwise, it solves the decision problem on its input. For this reason, decision problems L are also often called *languages*: we can view the world of all possible inputs $x \in \{0,1\}^*$ as being split into two subsets, the "yes" instances $x \in \mathsf{L}$ (input strings that belong to the language), and the "no" instances $x \notin \mathsf{L}$. By default, any $x \in \{0,1\}^*$ that is not a valid input to the problem automatically does not belong to L.

Karp vs. Cook reductions. There is one more subtlety related to reductions that we wish to comment on here. For two decision problems A and B, the definition we gave at the beginning of the section for $\mathsf{A} \leq_{\mathrm{P}} \mathsf{B}$ is often referred to in the literature as a *Cook reduction* from A to B.[2] In a Cook reduction, an algorithm for A can call an algorithm $\mathsf{AlgoForB}$ for B as many times as it would like (as long as the number is polynomial in the input size), and can perform whatever polytime computation it likes on top of the answers returned by $\mathsf{AlgoForB}$ to obtain its answer for A.

Interestingly, all reductions we discuss will be *Karp reductions*,[3] a specific type of Cook reduction. A Karp reduction from A to B has two special properties: it calls an algorithm $\mathsf{AlgoForB}$ for deciding B just once, and it immediately outputs the answer from $\mathsf{AlgoForB}$ as its own answer for deciding A. An equivalent way of viewing a Karp reduction is as a polytime transformation $f : \{0,1\}^n \to \{0,1\}^{\mathrm{poly}(n)}$ from $x = \langle a \rangle$ to $f(x) = \langle b \rangle$, where $\langle a \rangle$ is a length-n input to A, and $\langle b \rangle$ is an input to B. The transformation f preserves membership in the following sense.

- If $x = \langle a \rangle \in \mathsf{A}$, then $f(x) = \langle b \rangle \in \mathsf{B}$.
- If $x = \langle a \rangle \notin \mathsf{A}$, then $f(x) = \langle b \rangle \notin \mathsf{B}$.

In brief, a Karp reduction from A to B takes an input x, applies a polytime transformation f to it, and then returns the output of a decision algorithm for B on $f(x)$. This is just one (particularly simple) type of Cook reduction. In principle, one could design more complex Cook reductions from A to B that query $\mathsf{AlgoForB}$ multiple times and manipulate its outputs. In this course, we use $\mathsf{A} \leq_{\mathrm{P}} \mathsf{B}$ to just mean there exists some Cook reduction from A to B, whether it is Karp or not. However, it may help the reader to know that for all known reductions used to establish *NP-hardness* for natural problems, the central topic of Sections 3 and 4, Karp reductions are as powerful as Cook reductions in full generality (cf. Remark 1).

[2] Some authors use the terms *Turing reduction* or *oracle reduction* to describe a generalization of Cook reductions, where the number of oracle calls and additional computation time can be arbitrary (i.e., not necessarily polynomial).

[3] Analagously, the term *many-one reduction* is used to describe Karp reductions using more than polytime.

3 P vs. NP and Hardness of SAT

In this section, we introduce the P vs. NP problem, the most important unsolved problem in complexity theory. After defining the problem, we discuss consequences of either of the conclusions P = NP or P ≠ NP, by leveraging the notion of polytime reductions from Section 2, which we use to establish the central concept of NP-completeness. We conclude by defining the SAT decision problem, historically the first problem that was shown to be NP-complete.

3.1 P and NP

We begin this section by introducing P and NP, the two most well-studied complexity classes in complexity theory. Informally, a complexity class is a problem family sharing a common property in terms of their computational complexity, e.g., a bounded runtime. Throughout this section, let L be a *language*, a.k.a. *decision problem*; recall this means that L separates all binary inputs $x = \{0,1\}^*$ into "yes" instances $x \in \mathsf{L}$, and no instances $x \notin \mathsf{L}$.

Defining P. The complexity class P consists of all L decidable in polytime. Formally, $\mathsf{L} \in$ P if there is an algorithm $\mathcal{A}$ with runtime $\text{poly}(|x|)$ for any $x \in \{0,1\}^*$, deciding whether $x \in \mathsf{L}$. For brevity, in this case we simply say $\mathcal{A}$ is a polytime algorithm that decides L.

The reader should roughly think of P as containing all of the algorithmic problems we have studied thus far that admit polytime decision algorithms, with the important caveat that the runtime should be polynomial in the *input length*, so this does not include problems like SubsetSum or Knapsack (Section 3.3, Part III). To give a simple example, consider the language Reachability. An input $x \in \{0,1\}^*$ is in Reachability iff $x = \langle G, s, t \rangle$ encodes a graph $G = (V, E)$, and a pair of vertices $(s, t) \in V \times V$, such that s can reach t in G. This was the central problem of Sections 1 and 2 in Part V of the lecture notes. We can establish that Reachability ∈ P, i.e., that the language Reachability is decidable in polytime, using the following algorithm.

Algorithm 1: DecideReachability(x)

```
Input: x ∈ {0,1}*
if x is an encoding of ⟨G, s, t⟩ where G = (V, E) is a graph and (s, t) ∈ V × V then
    R ← GraphSearch(G, s)                          // Algorithm 1, Part V
    if t ∈ R then
        return True                                // t is reachable from s in G
    end
    return False                                   // x is a valid encoding, but t is not reachable from s in G
end
return False                                       // x is not a valid encoding
```

Recall from Section 2, Part V that GraphSearch is implementable in polytime, e.g., using a stack or a queue. It is thus straightforward to check that all steps of DecideReachability run in polytime, and moreover, it only returns **True** if x is a valid encoding $\langle G, s, t \rangle$, and s can reach t in G. Thus, DecideReachability meets all the criteria to establish that Reachability ∈ P.

Defining NP. The complexity class NP consists of all L decidable in *nondeterministic* polytime. Formally, $\mathsf{L} \in$ NP if there is a nondeterministic polytime algorithm $\mathcal{A}$ that decides L.

A nondeterministic algorithm $\mathcal{A}$ takes two inputs: $x \in \{0,1\}^*$, the original input for which we want to determine membership in L, and $h \in \{0,1\}^{\text{poly}(|x|)}$, an auxiliary "hint" or "witness." We say that the nondeterministic algorithm $\mathcal{A}$ decides L if it has the following properties.

- If $x \in \mathsf{L}$, then there is a choice of h such that $\mathcal{A}(x, h) =$ **True**.
- If $x \notin \mathsf{L}$, then for all choices of h, we have that $\mathcal{A}(x, h) =$ **False**.

Informally, NP can be viewed as the family of decision problems that admit succinct proofs. For languages L in NP, it may be difficult to decide for yourself whether a given input x is in L or not in polytime. However, if $\mathsf{L} \in$ NP, then every "yes" instance $x \in \mathsf{L}$ should be easy to *certify*: there should be a choice of the hint h that convinces our algorithm $\mathcal{A}$ that $x \in \mathsf{L}$ in polytime. Conversely, if $x \notin \mathsf{L}$, then no hint h should ever fool our decision algorithm into deciding that $x \in \mathsf{L}$.

The power of nondeterminism is strong, and indeed, many decision problems not known to be in P can very easily be seen to be in NP. Before giving some interpretations of this power, we begin with an example of such a language in NP. Consider the language VertexCover from Section 1. An input $x \in \{0,1\}^n$ is in VertexCover iff $x = \langle G, k \rangle$ encodes a graph $G = (V, E)$,[4] and a nonnegative integer $k \in \mathbb{Z}_{\geq 0}$, such that G has a *vertex cover* $S \subseteq V$ of size $|S| = k$. Recall that S is a vertex cover if every edge $e = (u, v) \in E$ either has $u \in S$ or $v \in S$.

It is not known how to decide VertexCover in polytime, so we do not know whether VertexCover $\in$ P. However, the following nondeterministic algorithm demonstrates VertexCover $\in$ NP.

Algorithm 2: DecideVertexCoverNP(x, h)

```
1  Input: x ∈ {0,1}*
2  if x is an encoding of ⟨G, k⟩ where G = (V, E) is a graph and k ∈ ℕ then
3      if h is an encoding of ⟨S⟩ where S ⊆ V and |S| = k then
4          for e = (u, v) ∈ E do
5              if u ∉ S and v ∉ S then
6                  return False                    // S is not a vertex cover
7              end
8          end
9          return True
10     end
11 end
12 return False                                    // x or h is not a valid encoding
```

Algorithm 2 asks the hint (auxiliary input) h to "spell out the answer" by providing an encoding of the vertex cover S itself as h. We can verify that it meets the criteria of a nondeterministic algorithm deciding L. Indeed, if $x \in \mathsf{L}$, then there is a valid vertex cover $S \subseteq V$ of size $|S| = k$, so we can provide $h = \langle S \rangle$ to the algorithm to cause it to return **True**. Conversely, if $x \notin \mathsf{L}$, then either it will be rejected on Line 12, or it encodes a graph without a size-k vertex cover. In the latter case, no choice of hint h will ever cause Algorithm 2 to return **True**.

Algorithm 2 may seem straightforward. For many NP languages, the corresponding nondeterministic decision algorithm will be just as simple. For example, the Maxcut decision problem accepts all $x = \langle G, k \rangle$ where graph $G = (V, E)$ has at least k edges crossing some cut $(S, V \setminus S)$. Again, Maxcut is in NP because we can provide $\langle S \rangle$ as the hint h, where S witnesses the maxcut in G. The nondeterministic decision algorithm simply counts the number of edges between S and $V \setminus S$, and verifies it is $\geq k$. NP also contains problems like Sudoku, where we should accept an input x if it encodes a partially-filled Sudoku game board that can be completed to a valid fully-filled game board. It is unknown how to find a Sudoku solution in polynomial time, so we do not know whether Sudoku $\in$ P. However, we can verify that a completed game board (encoded in h) is a valid solution and agrees with our partially-filled input, so Sudoku $\in$ NP.

One interpretation of nondeterminism is that the algorithm designer has access to an all-powerful oracle that is very good at certifying "yes" instances $x \in \mathsf{L}$, but the designer never wants to accidentally accept a "no" instance $x \notin \mathsf{L}$. Whenever a yes instance x is passed as input, the oracle can be queried for the best possible hint h that certifies x should be accepted.

Another interpretation imagines we generate h ourselves using coin flips. For $x \in \mathsf{L}$, it should be possible to get lucky and select an accepting hint h. On the other hand, for $x \notin \mathsf{L}$, no matter how the coins flip, our algorithm should reject. In brief, if we let $h \in \{0,1\}^{\text{poly}(|x|)}$ be a uniformly random advice string, the nondeterministic decision algorithm $\mathcal{A}$ for L should satisfy:

$$\Pr\left[\mathcal{A}(x, h) = \textbf{True}\right] \begin{cases} > 0 & x \in \mathsf{L} \\ = 0 & x \notin \mathsf{L} \end{cases}.$$

The power of nondeterminism thus lets an algorithm have access to the "best possible coin flips" that could convince it to accept an input x. This also explains the naming of the class NP: a

[4] Henceforth, all graphs defined in these notes will be unweighted and undirected.

deterministic algorithm uses no randomness, whereas a nondeterministic algorithm has access to randomness h that could potentially change its behavior on an input.

It may feel that nondeterministic algorithms access strong powers that deterministic counterparts cannot. It is thus immensely surprising that the following problem is still open.

Are P and NP the same complexity class?

In other words, does $\mathrm{P} = \mathrm{NP}$, or does $\mathrm{P} \neq \mathrm{NP}$? Observe that every $\mathsf{L} \in \mathrm{P}$ also satisfies $\mathsf{L} \in \mathrm{NP}$, because a nondeterministic algorithm can ignore the auxiliary hint and directly check membership in L; access to nondeterminism only makes an algorithm more powerful. The question is therefore whether we can prove that some language $\mathsf{L} \in \mathrm{NP}$ does not belong to P.

3.2 NP-hardness

Despite the fact that we do not know if $\mathrm{P} = \mathrm{NP}$, we can still say interesting things about the *conditional hardness* of certain languages, e.g., if we assume that $\mathrm{P} \neq \mathrm{NP}$, then some specific decision problem L cannot be solved in polytime. This owes in large part to the theory of NP-hardness, which formalizes the notion of bottleneck problems for NP.

We say that a language L is NP-hard if for all languages $\mathsf{L}' \in \mathrm{NP}$, we have that $\mathsf{L}' \leq_{\mathrm{P}} \mathsf{L}$. In other words, given access to an oracle for an NP-hard language L, we can solve any problem in NP in polytime. If an NP-hard language L further belongs to NP, then we say it is NP-complete. These notions are useful because they admit the following conditional hardness result.

Corollary 1. *If* $\mathrm{P} \neq \mathrm{NP}$*, then any* NP-hard *language* L *cannot be in* P.

Proof. We prove the contrapositive: if any language $\mathsf{L} \in \mathrm{P} \cap \text{NP-hard}$, then $\mathrm{P} = \mathrm{NP}$. To establish this, we deduce that any $\mathsf{L}' \in \mathrm{NP}$ is also in P. Indeed, because $\mathsf{L} \in \text{NP-hard}$ and $\mathsf{L}' \in \mathrm{NP}$, we know $\mathsf{L}' \leq_{\mathrm{P}} \mathsf{L}$ by the definition of NP-hard. Now by Lemma 1, $\mathsf{L} \in \mathrm{P}$ implies that $\mathsf{L}' \in \mathrm{P}$. □

Our use of Lemma 1 in Corollary 1 makes our notion of hardness more clear. Informally, we say that B is harder than a language A, if it is possible that $\mathsf{A} \in \mathrm{P}$ but $\mathsf{B} \notin \mathrm{P}$, but not vice versa. This is the case if $\mathsf{A} \leq_{\mathrm{P}} \mathsf{B}$, by Lemma 1. From this perspective, any NP-hard language L is harder than any language $\mathsf{L}' \in \mathrm{NP}$, because we know $\mathsf{L} \in \mathrm{P} \implies \mathsf{L}' \in \mathrm{P}$, but not vice versa. It follows that NP-complete consists of the hardest problems within NP, and that to prove $\mathrm{P} = \mathrm{NP}$, it suffices to give a polytime decision algorithm for *any* NP-complete language.

It is helpful to think of the theory of polytime reductions and hardness as deriving consequences based on the outcome of the P vs. NP conjecture. We do not know whether we live in a world where $\mathrm{P} = \mathrm{NP}$ (colloquially called "Algorithmica"), or a world where $\mathrm{P} \neq \mathrm{NP}$ (colloquially called "Pessiland").[5] However, we can say something about what those worlds might look like.

Life in Pessiland. If $\mathrm{P} \neq \mathrm{NP}$, then by Corollary 1, we can conclude that we should stop trying to design polytime algorithms for any NP-hard problem — there simply are not any! As we will see in Section 4, this is powerful because an incredibly broad range of basic algorithmic problems are provably NP-hard. This includes the SAT decision problem (discussed in Section 3.3), as well as other problems we have mentioned, e.g., VertexCover, SubsetSum, Maxcut, and Sudoku. It also includes the famous traveling salesman problem from operations research. We refer the reader to Chapter 12, [Eri24], for a longer list of well-studied NP-hard problems.

Although we are not sure whether $\mathrm{P} \neq \mathrm{NP}$ is true, a survey conducted in 2019 [Hem19] showed that $\approx$ 88-99% of experts believed $\mathrm{P} \neq \mathrm{NP}$. While too much stock should not be placed on such subjective claims, it does suggest where we should devote our research efforts. We either must directly show that $\mathrm{P} = \mathrm{NP}$ (i.e., that the aforementioned experts are wrong), or we should give up on polytime algorithms for all NP-complete problems and focus on alternatives.

Life in Algorithmica. On the other hand, if $\mathrm{P} = \mathrm{NP}$, then because every language in NP reduces to SAT, designing faster SAT algorithms can yield algorithms for a whole host of other problems.

[5] This dichotomy is adapted from a famous paper of Russell Impagliazzo [Imp95], who actually defines five possible worlds delineating different scenarios under which $\mathrm{P} \neq \mathrm{NP}$, and what sorts of cryptographic primitives we can derive as a consequence. For simplicity in these lecture notes, we only define two worlds.

This is analogous in spirit to maxflow reductions (Section 5.3, Part V), which focus our attention on designing faster maxflow algorithms. Unfortunately, life in Algorithmica is bad news for many primitives in cryptography, which rely on hard computational problems that are easy to solve with a "secret key" (e.g., the hint in a nondeterministic algorithm).

3.3 Cook-Levin theorem

How do we show that a language L is NP-hard? The starting point is the following seminal result due to Cook [Coo71] and Levin [Lev73].

Theorem 1 (Cook-Levin theorem)**.** *The language* SAT *is* NP-hard.

The most remarkable thing about Theorem 1 is that it is unconditional: it is true whether or not P = NP. Indeed, we will see that $\mathsf{SAT} \in$ NP, so Theorem 1 formally establishes that SAT is the hardest problem in NP in the sense of polytime reductions. Of course, if P = NP, then it is not so impressive that every $\mathsf{L}' \in$ NP has a polytime reduction to SAT, because then L' can be solved in polytime without even querying a SAT oracle! The utility of Theorem 1 is that it clarifies the "all-or-nothing" nature of the P vs. NP conjecture for a concrete bottleneck problem: either $\mathsf{SAT} \in$ P (in which case P = NP), or $\mathsf{SAT} \notin$ P (in which case P $\neq$ NP).

We remark that SAT is not the *unique* hardest problem in NP, as there are other NP-complete problems that also satisfy this all-or-nothing phenomenon. Indeed, Section 4 contains several such examples. However, SAT was historically the first problem shown to be NP-complete, and the hardness of other problems in Section 4 is based on SAT. In the rest of the section, we simply define SAT, and provide some intuition for why Theorem 1 is true.

Defining SAT. Consider n Boolean *variables* $\{x_1, x_2, \ldots, x_n\}$, each of which is either **True** or **False**. We use $\neg v$ to denote the negation of a variable v, i.e., if $v =$ **True** then $\neg v =$ **False** and vice versa. We call any variable or its negation a *literal*, so $x_1, \neg x_1, \ldots, x_n, \neg x_n$ are all literals.

A Boolean formula is any expression formed by combining literals, using the operations of $\wedge$ (shorthand for **and**) and $\vee$ (shorthand for **or**). A Boolean formula in *conjunctive normal form* (a.k.a. a CNF formula) has a special form: it is an **and** of multiple clauses $\phi_1, \phi_2, \ldots, \phi_m$, where each clause is an **or** of literals.[6] Below we provide an example CNF formula in three variables:

$$\Phi = \underbrace{(x_1)}_{\phi_1} \wedge \underbrace{(\neg x_1 \vee x_2)}_{\phi_2} \wedge \underbrace{(\neg x_1 \vee \neg x_2 \vee x_3)}_{\phi_3} \wedge \underbrace{(\neg x_3)}_{\phi_4}. \tag{1}$$

We call a CNF formula *satisfiable* if there exists an assignment of **True** or **False** values to the relevant variables, such that the overall formula evaluates to **True**. Due to the special form of CNF formulas as an **and** over clauses, an equivalent criterion is that at least one literal in every clause must be **True**. If Φ is a CNF formula with at least one satisfying assignment, then $\langle \Phi \rangle \in \mathsf{SAT}$; conversely, any input that does not encode a satisfiable CNF formula is not in SAT.

It is not hard to see that Φ in (1) is an *unsatisfiable* CNF formula: the first clause ϕ_1 forces $x_1 =$ **True**, ϕ_2 then forces $x_2 =$ **True**, ϕ_3 then forces $x_3 =$ **True**, and ϕ_4 then evaluates to **False**. On the other hand, if we changed the last clause ϕ_4 to $\phi_4' := (x_3)$ to produce a new CNF formula Φ', then the assignment $\{x_1 =$ **True**$, x_2 =$ **True**$, x_3 =$ **True**$\}$ causes Φ' to evaluate to **True**. Thus, Φ' has a satisfying assignment, and so $\langle \Phi' \rangle \in \mathsf{SAT}$.

We observe that $\mathsf{SAT} \in$ NP, because a nondeterministic algorithm can request a witness h encoding a satisfying assignment to an input formula $x = \langle \Phi \rangle$, and check that Φ is indeed satisfiable. In light of Theorem 1, this shows $\mathsf{SAT} \in$ NP-complete, our first example of such a problem.

Proof sketch of Theorem 1. The proof of Theorem 1 goes beyond the scope of this course, but we provide a rough sketch of one known proof here, deferring more formal details (and alternative proofs) to the excellent expositions in Sections 2.3 and 6.1 of [AB09].

The goal of Theorem 1 is to show that every $\mathsf{L} \in$ NP satisfies $\mathsf{L} \leq_{\mathrm{P}} \mathsf{SAT}$. What this means is that we wish to transform any possible input x into a CNF formula $\langle \Phi \rangle = f(x)$, such that $f(x) \in \mathsf{SAT}$ iff

[6] We only consider CNF formulas without loss of generality: a procedure known as the Tseytin transformation converts any Boolean formula into an equivalent CNF formula with only a constant factor blowup.

$x \in \mathsf{L}$. The basic idea is to use the existence of a nondeterministic algorithm $\mathcal{A}(x, w)$ that decides whether $x \in \mathsf{L}$, and define the transformation $x \to f(x) = \langle \Phi \rangle$ by using $\mathcal{A}$.

More concretely, suppose that $\mathcal{A}$ takes as input a Boolean string $x \in \{0,1\}^n$, and wants to determine whether $x \in \mathsf{L}$ by using an auxiliary input $h \in \{0,1\}^{g(n)}$, where the hint length $g(n) = \text{poly}(n)$. We can view the operations performed by $\mathcal{A}$ as implicitly defining a "circuit" $\mathcal{C}$ consisting of **and**, **or**, and **not** gates that map the input bits in x and h to an output, either **True** or **False**. This is because combinations of the operations **and**, **or**, and **not** are expressive enough to capture any function of one or two bits. It is straightforward to see that the size of the circuit is a polynomial in n, because the total number of bit operations performed by $\mathcal{A}$ is $\text{poly}(n)$.

The circuit $\mathcal{C}$ has the following property: if $x \in \mathsf{L}$, then there is a choice of h that causes $\mathcal{C}$ to evaluate to **True**, and otherwise there is no such h. Imagine now that we compile the circuit $\mathcal{C}$ into an equivalent CNF formula Φ, where the variables to Φ are the bits in h, and the bits of the original input x are "hard-coded" into Φ. Then an equivalent statement is that Φ is satisfiable (i.e., there is a choice of h that causes Φ to evaluate to **True**) iff $x \in \mathsf{L}$. Thus, an algorithm that can decide whether $\langle \Phi \rangle \in \mathsf{SAT}$ can be used to determine whether $x \in \mathsf{L}$ in polynomial time. This (informal) reduction is exactly what we need to demonstrate that $\mathsf{L} \leq_{\text{P}} \text{NP}$. Of course, there are many details that need to be filled in to show that any nondeterministic algorithm can be converted into an equivalent circuit with "hard-coded" input bits.

There are various other subtleties with Theorem 1, including specifying the actual computational model we are working in, and the actual size of the blowup from the original input x to the SAT formula encoding $f(x)$. Exciting, a line of research has shown that *quasi-linear* Cook-Levin theorems [Sch78, Coo88, Rob91] hold in essentially every common computational model, such as multi-tape Turing machines and RAM. These are reductions from $\mathsf{L} \in \text{NP}$ to SAT with the following property: if there is a nondeterministic algorithm $\mathcal{A}$ that decides L in time $T(n)$, then there is an encoding $f(x) = \langle \Phi \rangle$ such that $x \in \mathsf{L}$ iff $f(x) \in \mathsf{SAT}$, and the size of Φ is $O(T(n) \log T(n))$. This is nearly the best blowup one could hope for in the Cook-Levin theorem.

4 NP-completeness

In this section, we establish that several widely-studied languages beyond SAT are also NP-complete: the $\mathsf{3SAT}$, $\mathsf{IndependentSet}$, Clique, $\mathsf{VertexCover}$, and $\mathsf{SubsetSum}$ decision problems. As we will see, it is straightforward to show that all of these problems are in NP; in each case, we can provide an answer as a certificate, and verify correctness in polynomial time.

The challenge will be in showing that these problems are also NP-hard. Our strategy will be to combine Theorem 1 (which gives us an initial NP-hard language) with the following observation.

Lemma 2. *If* $\mathsf{A} \in \text{NP-hard}$*, and* $\mathsf{A} \leq_{\text{P}} \mathsf{B}$*, then* $\mathsf{B} \in \text{NP-hard}$ *as well.*

Proof. We want to show a polytime reduction from L to B, where $\mathsf{L} \in \text{NP}$ is arbitrary. We know there are polytime reductions from L to A (because $\mathsf{A} \in \text{NP-hard}$) and A to B (by assumption). Composing these reductions, i.e., replacing every oracle call to A with polynomial work and polynomially many oracle calls to B, yields the desired polytime reduction from L to B. □

Lemma 2 is rather intuitive: recall that a language A is in NP-hard if, given an oracle for A, we can decide all of NP in polytime. The assumption $\mathsf{A} \leq_{\text{P}} \mathsf{B}$ means that an oracle for B can simulate an oracle for A in polytime, so B is also powerful enough to decide NP in polytime.

Thus, if we want to show a language L is NP-hard, it is enough to show $\mathsf{SAT} \leq_{\text{P}} \mathsf{L}$ and then apply Lemma 2 and Theorem 1. Of course, once we have shown more languages are NP-hard via this strategy, we can instead use them as the "base" problem A when we apply Lemma 2. We give a range of such examples, yielding many more base NP-hard problems for future use.

4.1 3SAT

In this section, we show that $\mathsf{3SAT} \in \text{NP-complete}$. The $\mathsf{3SAT}$ decision problem is defined as follows: an input x is in $\mathsf{3SAT}$ iff $x = \langle \Phi \rangle$ encodes a CNF formula (cf. Section 3.3) that is both

satisfiable, and has *exactly three* literals per clause. We call such a CNF formula a 3CNF formula for short. For example, an encoding $\langle \Phi \rangle$ of the 3CNF formula

$$\Phi = \underbrace{(x_1 \vee x_2 \vee \neg x_3)}_{\phi_1} \wedge \underbrace{(x_2 \vee x_3 \vee \neg x_4)}_{\phi_2}$$

belongs to 3SAT, as witnessed by the assignment $\{x_1 = \textbf{True}, x_2 = \textbf{True}, x_3 = \textbf{True}, x_4 = \textbf{True}\}$. It is clear that 3SAT $\in$ NP, for the same reason that SAT $\in$ NP: we can provide the satisfying assignment as the hint h, and verify it in polynomial time.

Proposition 1. *The language* 3SAT *is* NP-hard.

Proof. We show that SAT $\leq_{\mathrm{P}}$ 3SAT, and then apply Lemma 2 and Theorem 1.

Let $\Phi = \phi_1 \vee \phi_2 \vee \ldots \vee \phi_m$ be a CNF formula. We transform Φ into a 3CNF formula $\widetilde{\Phi} = f(\Phi)$, such that $\langle \widetilde{\Phi} \rangle \in$ 3SAT iff $\langle \Phi \rangle \in$ SAT, and $\langle \widetilde{\Phi} \rangle$ is polynomially larger than $\langle \Phi \rangle$. Accomplishing this shows that SAT $\leq_{\mathrm{P}}$ 3SAT: our algorithm for SAT simply computes $\widetilde{\Phi} = f(\Phi)$ and calls a 3SAT oracle on $\widetilde{\Phi}$. Note that we have described a Karp reduction from SAT to 3SAT (cf. Section 2).

We now describe our transformation from Φ to $\widetilde{\Phi}$. We proceed clause by clause. If any clause ϕ in Φ has 1, 2, or 3 literals, it is easy to form a single equivalent 3-literal clause $\widetilde{\phi}$ in $\widetilde{\Phi}$:

$$(\ell_1) \equiv (\ell_1 \vee \ell_1 \vee \ell_1), \quad (\ell_1 \vee \ell_2) \equiv (\ell_1 \vee \ell_1 \vee \ell_2), \quad (\ell_1 \vee \ell_2 \vee \ell_3) \equiv (\ell_1 \vee \ell_2 \vee \ell_3).$$

It remains to transform clauses with ≥ 4 literals. We replace any such clause $\phi = \ell_1 \vee \ell_2 \vee \ldots \ell_k$ in Φ with a conjunction of $k-2$ clauses in $\widetilde{\Phi}$, each of length 3. Specifically, for each such length-$k \geq 4$ clause ϕ in Φ, we introduce $k-3$ new variables $z_1, z_2, \ldots, z_{k-3}$, and replace ϕ with the clauses

$$\begin{aligned} \phi \equiv & \underbrace{(\ell_1 \vee \ell_2 \vee z_1)}_{\widetilde{\phi}_1} \wedge \underbrace{(\neg z_1 \vee \ell_3 \vee z_2)}_{\widetilde{\phi}_2} \wedge \underbrace{(\neg z_2 \vee \ell_4 \vee z_3)}_{\widetilde{\phi}_3} \\ & \ldots \wedge \underbrace{(\neg z_{k-4} \vee \ell_{k-2} \vee z_{k-3})}_{\widetilde{\phi}_{k-3}} \wedge \underbrace{(\neg z_{k-3} \vee \ell_{k-1} \vee \ell_k)}_{\widetilde{\phi}_{k-2}}. \end{aligned} \tag{2}$$

For example, if $k = 5$, we would replace $\phi = (\ell_1 \vee \ell_2 \vee \ell_3 \vee \ell_4 \vee \ell_5)$ in Φ with the three clauses

$$(\ell_1 \vee \ell_2 \vee z_1) \wedge (\neg z_1 \vee \ell_3 \vee z_2) \wedge (\neg z_2 \vee \ell_4 \vee \ell_5)$$

in $\widetilde{\Phi}$, creating two new variables z_1, z_2 for this purpose.

Let us follow notation in (2). Our key claim is that there is an assignment to $\{z_1, z_2, \ldots, z_{k-3}\}$ such that $\widetilde{\phi}_1 \vee \widetilde{\phi}_2 \vee \ldots \widetilde{\phi}_{k-2} = \textbf{True}$, iff at least one of $\{\ell_1, \ell_2, \ldots, \ell_k\}$ is already **True**.

To see this, suppose first that all of the original literals $\{\ell_1, \ell_2, \ldots, \ell_k\}$ were **False**. Then to make $\widetilde{\phi}_1 = \textbf{True}$, we must set $z_1 \leftarrow \textbf{True}$; $\widetilde{\phi}_2$ then forces $z_2 \leftarrow \textbf{True}$, and continuing similarly, all of the $\{z_1, z_2, \ldots, z_{k-3}\}$ are forced to be **True** by the clauses $\widetilde{\phi}_1, \widetilde{\phi}_2, \ldots, \widetilde{\phi}_{k-3}$. However, this assignment violates $\widetilde{\phi}_{k-2}$, as it is now the conjunction of three **False** literals.

On the other hand, if even a single one of the original literals $\{\ell_1, \ell_2, \ldots, \ell_k\}$ is **True**, there is a way to satisfy all of $\widetilde{\phi}_1, \widetilde{\phi}_2, \ldots, \widetilde{\phi}_{k-2}$. Indeed, if $\ell_i = \textbf{True}$, by setting $z_1 = z_2 = \ldots = z_{i-2} = \textbf{True}$, and $z_{i-1} = \ldots = z_{k-2} = \textbf{False}$, we can check that all of $\widetilde{\phi}_1, \widetilde{\phi}_2, \ldots, \widetilde{\phi}_{k-2}$ are satisfied.

We now complete the proof that the original CNF formula Φ is satisfiable iff the transformed 3CNF formula $\widetilde{\Phi}$ is. If Φ is satisfiable, take some satisfying assignment of its literals. By our observation, we can extend this assignment to an assignment of the new variables in $\widetilde{\Phi}$, so that every set of new clauses $\widetilde{\phi}_1, \widetilde{\phi}_2, \ldots, \widetilde{\phi}_{k-2}$ created out of some clause ϕ in Φ is also satisfied. Conversely, if every assignment of original literals in Φ fails to satisfy some clause, then take ϕ to be any unsatisfied clause in Φ. No assignment of new variables can suddenly cause the new clauses $\widetilde{\phi}_1, \widetilde{\phi}_2, \ldots, \widetilde{\phi}_{k-2}$ associated with ϕ to be satisfied, so we can conclude that $\widetilde{\Phi}$ is also unsatisfiable. □

Proposition 1 is useful because it gives us a more structured problem to reduce from (3SAT), as a starting point to show that other decision problems are NP-hard. We will capitalize on the additional structure afforded by 3SAT in the following Section 4.2.

Perhaps the most mysterious thing about Proposition 1 is the magic number 3. This choice is for good reason: 2SAT (deciding whether a 2CNF formula is satisfiable) turns out to be solvable in polynomial time, so 3SAT is the simplest structured SAT problem that is NP-hard. Intuitively, this is because a 3CNF clause $(\ell_1 \vee \ell_2 \vee \ell_3)$ induces *branching*: fixing $\ell_1 \leftarrow$ **False** still requires making a decision about which of ℓ_2 or ℓ_3 to set to **True**, an issue that does not arise in 2SAT.

4.2 Independent set and relatives

In this section, we show that a basic graph problem, IndependentSet, is also NP-complete; as a corollary, we conclude that the related problems Clique and VertexCover are also NP-complete. This example demonstrates the ingenuity that can go into NP-hard reductions: perhaps surprisingly, we reduce from a Boolean satisfiability problem (3SAT) to a graph problem (IndependentSet).

The IndependentSet decision problem is defined as follows: an input x is in IndependentSet iff $x = \langle G, k \rangle$ encodes a graph $G = (V, E)$, and a positive integer $k \in \mathbb{N}$, such that G has an *independent set* $S \subseteq V$ of size k. Here, we call a set of vertices S independent if there are no edges between any pair of vertices in S, i.e., for all $(u, v) \in E$, either $u \notin S$ or $v \notin S$.

It is again immediate that IndependentSet $\in$ NP, because the witness h can encode a set S, and a nondeterministic algorithm can verify that $|S| = k$ and that S is in fact independent.

Proposition 2. *The language* IndependentSet *is* NP-hard.

Proof. We show that 3SAT $\leq_{\mathrm{P}}$ IndependentSet, and then apply Lemma 2 and Proposition 1.

Let $\Phi = \phi_1 \vee \phi_2 \vee \ldots \vee \phi_m$ be a 3CNF formula. We transform Φ into a graph G on $3m$ vertices, such that $\langle G, m \rangle \in$ IndependentSet iff $\langle \Phi \rangle \in$ 3SAT. As in Proposition 1, such a Karp reduction implies that 3SAT $\leq_{\mathrm{P}}$ IndependentSet. We now describe how to construct G from Φ in two steps.

First, for each clause $\phi = (\ell_1 \vee \ell_2 \vee \ell_3)$ in Φ, we create a complete graph on three new vertices (i.e., a triangle) in G, and label these vertices with ℓ_1, ℓ_2, and ℓ_3. Note that a single literal may participate in multiple clauses; in this case, we create a new vertex labeled with the literal for each clause it participates in. In total, this step creates $3m$ vertices and $3m$ edges in G.

Second, consider each variable v such that both v and $\neg v$ appear as literals in Φ. We add a perfect matching between all copies of v and $\neg v$ in G, for each such variable v; that is, we add an edge between every copy of v and every copy of $\neg v$. For example, if v appears in 3 clauses and $\neg v$ appears in 2, then this step adds a total of 6 edges to G due to v.

We claim that G admits an independent set S of size m iff Φ is satisfiable. Note that any such independent set must contain exactly one vertex from each of the m triangles in G, because every pair of vertices within the same triangle is joined by an edge.

To see one direction, suppose that Φ is satisfiable. Consider any satisfying assignment, and select exactly one **True** literal per clause in Φ (if there are multiple in a clause, choose one arbitrarily). We claim that taking S to be the set of vertices corresponding to these literals in G is an independent set of size m. To show independence, none of the "triangle edges" added in the first step of our construction go between vertices in S, as every vertex lies in a unique triangle. Similarly, none of the "matching edges" added in the second step go between vertices in S, because we can never pick both a variable v and its negation $\neg v$, as they cannot both be **True** literals.

To see the other direction, suppose that G admits a vertex cover S of size m. As argued previously, S must select exactly one literal per triangle; set the corresponding literals in Φ to be **True**, and set their negations to be **False**. Note that this never causes any inconsistencies, because a variable v and its negation $\neg v$ cannot both belong to S due to the presence of the matching edges. Finally, if a variable v and its negation $\neg v$ both do not belong to S, we can assign its value arbitrarily in Φ. This assignment satisfies Φ, as every clause has at least one **True** literal. □

We have thus shown that IndependentSet is NP-complete. This is particularly convenient for proving that other graph-structured problems are NP-complete, as reducing from IndependentSet rather than a satisfiability problem leads to somewhat simpler constructions. We conclude the section with two such reductions from IndependentSet to related graph problems.

Clique. As an example, we use Proposition 2 to deduce that Clique is also NP-complete.

The Clique decision problem is defined as follows: an input x is in IndependentSet iff $x = \langle G, k \rangle$ encodes a graph $G = (V, E)$, and a positive integer $k \in \mathbb{N}$, such that G has a *clique* $S \subseteq V$ of size k. Here, we call a set of vertices S a clique (a.k.a. complete graph) if every pair of vertices in S has an edge between them, i.e., for all $(u, v) \in S \times S$, we have $(u, v) \in E$. Clearly, Clique $\in$ NP because we can encode the relevant set S in the witness h.

Corollary 2. *The language* Clique *is* NP-hard.

Proof. We show that IndependentSet $\leq_{\mathrm{P}}$ Clique, and then apply Lemma 2 and Proposition 2.

Our reduction is very simple. Let $\langle G, k \rangle$ be an input to IndependentSet, where $G = (V, E)$ is the graph in question. Define an "inverse graph" $H = (V, F)$ that flips the presence of edges in G as follows: if $(u, v) \in V \times V$ is an edge in G, then do not add (u, v) to F, and if $(u, v) \notin E$, then add (u, v) to F. Every independent set in G is then a clique in H of the same size. Thus, there is an independent set of size k in G iff there is a clique of size k in H.

In conclusion, $\langle H, k \rangle \in$ Clique iff $\langle G, k \rangle \in$ IndependentSet. This is a Karp reduction that demonstrates IndependentSet $\leq_{\mathrm{P}}$ Clique as desired. □

Vertex cover. As another example, we deduce that VertexCover (defined in Section 2) is also NP-complete. Recall that we already showed using Algorithm 2 that VertexCover $\in$ NP.

To establish our reduction from IndependentSet to VertexCover, we need a helper fact.

Lemma 3. *If $G = (V, E)$ is a graph, $S \subseteq V$ is an independent set iff $V \setminus S$ is a vertex cover.*

Proof. To see one direction, assume that $S \subseteq V$ is an independent set, and suppose for contradiction that some edge $e = (u, v) \in E$ is not covered by $V \setminus S$ so that it is not a vertex cover. This means that neither u nor v lies in $V \setminus S$, i.e., both vertices are in S. This contradicts the fact that S is an independent set, because e goes between two of its vertices.

To see the other direction, suppose that $V \setminus S$ is a vertex cover, but that there is an edge e going between two vertices in S. This is a contradiction, because $V \setminus S$ does not cover e. □

Corollary 3. *The language* VertexCover *is* NP-hard.

Proof. We show that IndependentSet $\leq_{\mathrm{P}}$ VertexCover, and then apply Lemma 2 and Proposition 2.

Our reduction is again very simple. Let $\langle G, k \rangle$ be an input to IndependentSet, where $G = (V, E)$ is the graph in question. By Lemma 3, G has an independent set of size k iff it has a vertex cover of size $|V| - k$. Hence, $\langle G, |V| - k \rangle \in$ VertexCover iff $\langle G, k \rangle \in$ IndependentSet. This Karp reduction implies our claim IndependentSet $\leq_{\mathrm{P}}$ VertexCover. □

4.3 Subset sum

Our final NP-complete problem in these notes is SubsetSum, as studied in Section 3.3, Part III. As remarked in Section 1, no algorithm is known whose runtime is polynomial in the input size (particularly for exponentially-sized target values, for which DP fails).

We briefly redefine SubsetSum as a decision problem. An input x is in SubsetSum iff $x = \langle A, T \rangle$ encodes an Array of positive integers $\{a_i \in \mathbb{N}\}_{i \in [m]}$, and a target value $T \in \mathbb{N}$, such that there exists a subset $S \subseteq [m]$ satisfying $\sum_{i \in [m]} a_i = T$, i.e., T is realized as a subset sum.

It is straightforward to see that SubsetSum $\in$ NP. A nondeterministic decision algorithm can accept an encoding of the realizing set S as its witness h, and verify that $\sum_{i \in [m]} a_i = T$ in polynomial time. The key challenge is to find an NP-hard problem to reduce from; so far, we have only discussed Boolean satisfiability problems and graph problems, neither of which have an obvious connection to the numerical flavor of SubsetSum. For this reason the author finds the following reduction, a variant of which is first found in [Kar72],[7] to be particularly ingenious.

[7] This seminal paper defined a set of 21 problems, including SAT, and derived reductions between them to show that all 21 are NP-complete, by building upon Theorem 1 and using Lemma 2.

Proposition 3. *The language* SubsetSum *is* NP-hard.

Proof. We show that VertexCover $\leq_P$ SubsetSum, and then apply Lemma 2 and Corollary 3.

Let $G = (V, E)$ be a graph, and let $k \in \mathbb{N}$. We write $n := |V|$ and $m := |E|$ for short. Our goal is to transform G into a set A of positive integers, and a target value $T \in \mathbb{N}$, such that $\langle G, k\rangle \in$ VertexCover iff $\langle A, T\rangle \in$ SubsetSum, in polytime and with at most polynomial blowup in input size. This implies the desired VertexCover $\leq_P$ SubsetSum.

Our construction adds one positive integer to A for each edge $e \in E$, and one positive integer for each vertex $v \in V$. We begin with the edges. Arbitrarily label the edges of E with the integers $0 \leq i \leq m-1$. For each $i \in [m-1]$, we add the integer $a_i := 4^i$ to A.

Next, for each vertex $v \in V$, let $\Delta(v) \subseteq E$ be all edges that v is an endpoint of. If we identify $\Delta(v)$ with a subset of $\{i \in \mathbb{Z} \mid 0 \leq i \leq m-1\}$, our edge labels, then we further add all of the integers

$$b_v := 4^m + \sum_{i \in \Delta(v)} 4^i \tag{3}$$

to the set A. Thus, the set A consists of all of the $\{a_i = 4^i\}_{0 \leq i \leq m-1}$, and all of the $\{b_v\}_{v \in V}$ defined in (3). We finally set the target value to be the integer $T := k \cdot 4^m + \sum_{i=0}^{m-1} 2 \cdot 4^i$.

In total, $\langle A, T\rangle$ can clearly be constructed in polytime with polynomial blowup in size.

We claim that there is a subset of A with sum T, iff there is a vertex cover of size k in G. To show this, it is helpful to write all of the integers $\{a_i\}_{0 \leq i \leq m-1}$, $\{b_v\}_{v \in V}$, and T, in base 4.

- Each a_i has a single 1 in the i^{th} place, and 0s elsewhere.
- Each b_v defined in (3) has a 1 in the m^{th} place, along with a 1 in the i^{th} place for every edge $0 \leq i \leq m-1$ that v is an endpoint of. All other places are 0s.
- T has a 2 in the i^{th} place for all $0 \leq i \leq m-1$, and the leading order places have k in base-4.

For example, if we want to check that the complete graph on 4 nodes and 6 edges has a vertex cover of size 1, we create the SubsetSum instance (with all numbers in base 4):

$$\{a_i\}_{0 \leq i \leq 5} = \{\underbrace{0000001_4}_{\text{edge (1,2)}}, \underbrace{0000010_4}_{\text{edge (1,3)}}, \underbrace{0000100_4}_{\text{edge (1,4)}}, \underbrace{0001000_4}_{\text{edge (2,3)}}, \underbrace{0010000_4}_{\text{edge (2,4)}}, \underbrace{0100000_4}_{\text{edge (3,4)}}\},$$

$$\{b_v\}_{v \in V} = \{\underbrace{1000111_4}_{\text{vertex 1}}, \underbrace{1011010_4}_{\text{vertex 2}}, \underbrace{1101010_4}_{\text{vertex 3}}, \underbrace{1110100_4}_{\text{vertex 4}}\}, \quad T = 1222222_4.$$

The key observation is that any subset sum of A has no "overflow" in any of the places $0 \leq i \leq m-1$. Notice that for any $0 \leq i \leq m-1$, there are exactly 3 elements of A that have a nonzero i^{th} place. This is because if we associated the label i with the edge $e = (u, v) \in E$, then only a_i, b_u, and b_v have a 1 in their i^{th} places! As no digit sum can exceed 3, overflow cannot occur.

Now consider how we could achieve a target sum of T with a subset of S. We must select exactly k of the vertex values $\{b_v\}_{v \in V}$ to achieve a leading-order value of $k \cdot 4^m$. These values have a sum (in base 4) that has 0, 1, or 2 in the i^{th} place, for all $0 \leq i \leq m-1$.

We claim that, when summing up the contributions of the chosen subset of $\{b_v\}_{v \in V}$, if the i^{th} place is a 0 for any $0 \leq i \leq m-1$, then it is impossible for our subset sum to equal T. Indeed, the only other values we could add to our subset sum are from the $\{a_i\}_{0 \leq i \leq m-1}$, and these cannot increase the i^{th} place enough to reach a 2, as required by T.

Thus, the only way we can achieve T is if some subset of vertices $S \subseteq V$ is chosen, satisfying $|S| = k$, and that $\sum_{v \in V} b_v$ has a 1 or 2 in its i^{th} place for all $0 \leq i \leq m-1$ when written in base 4. This means that the subset S covers every edge, because only endpoints of the i^{th} edge induce values that contribute to the i^{th} place, i.e., S is a vertex cover of size k.

To complete our reduction, we must show that any vertex cover in G of size k induces a subset of A that sums to exactly T. To do so, first sum the subset of the $\{b_v\}_{v \in V}$ corresponding to the vertex cover, and then add the a_i for any i such that the i^{th} place currently has a 1 as its value. □

5 Beyond P and NP

Complexity theory is broad, and encompasses many other interesting classes beyond P and NP. In this section, we give a basic primer on some of the most important such classes.

5.1 Computability

Thus far, our discussion has been centered around three complexity classes: P, NP, and NP-hard. We have observed that $\text{P} \subseteq \text{NP}$, and that $\text{NP} \cap \text{NP-hard} = \text{NP-complete}$ is nonempty (regardless of whether $\text{P} = \text{NP}$). While it is unclear if $\text{NP} \setminus \text{P}$ is nonempty, we can definitively say that there exist languages in $\text{NP-hard} \setminus \text{NP}$, demonstrating the limitations of NP.

Trading nondeterminism for time. Recall that a language L is simply a subset of all possible binary strings $\{0,1\}^*$. We say that an algorithm $\mathcal{A}$ decides L if it can take input $x \in \{0,1\}^*$ and determine whether $x \in \mathsf{L}$ or not. Some languages are decidable in polytime ($\mathsf{L} \in \text{P}$), possibly with the help of nondeterminism ($\mathsf{L} \in \text{NP}$). What might a language look like that is not even in NP?

Let us begin by giving a more refined definition of time complexity. For any function $f : \mathbb{N} \to \mathbb{N}$, let $\text{TIME}(f(n))$ be the class of languages L, such that there is an algorithm $\mathcal{A}$ that decides whether a length-n input $x \in \{0,1\}^n$ belongs to L within $f(n)$ time. For brevity, we may lump in multiple functions using asymptotics, e.g., $\text{TIME}(O(n^2))$ refers to all languages decidable in quadratic time (without specifying what quadratic). Observe that $\text{TIME}(\text{poly}(n)) = \text{P}$.

One caveat is that we must be careful with specifying what it means for an algorithm to run in time $f(n)$: a slow processor could yield a constant factor overhead, and even the types of allowed operations, e.g., moving a pointer to adjacent memory cells only vs. "random access" memory (RAM), can make a difference. For simplicity, we fix a model of computation, and measure all time bounds with respect to that model. A standard model is *multi-tape Turing machines*, which place the input on a read-only tape, the output on a write-only tape, and have work tapes to perform intermediate computations. On multi-tape Turing machines, a unit of time is one operation, e.g., reading one bit, writing one bit, or moving to an adjacent cell on any tape.

Graciously, as far as we know,[8] multi-tape Turing machines can simulate essentially any other model of computation with polynomial overhead, a variant of the "Church-Turing thesis." Thus, for the purposes of determining which languages can be decided in polynomial time, the multi-tape Turing machine is a perfectly fine model to stick with when defining $\text{P} = \text{TIME}(\text{poly}(n))$.

We next note that NP is also provably contained in a time-bounded complexity class. The flipside of this result is that any language that takes too long to decide cannot be in NP.

Lemma 4. $\text{NP} \subseteq \text{TIME}(\exp(\text{poly}(n)))$.

Proof. Our goal is to decide any language $\mathsf{L} \in \text{NP}$ using a (deterministic) algorithm in $\exp(\text{poly}(n))$ time. Because $\mathsf{L} \in \text{NP}$, we know there is a nondeterministic algorithm $\mathcal{A}$ that decides L by accepting (x, w) for any length-n input $x \in \mathsf{L}$, and some choice of $w \in \{0,1\}^{\text{poly}(n)}$. We can simply create an algorithm $\mathcal{A}'(x)$ that loops over every possible $w \in \{0,1\}^{\text{poly}(n)}$ itself, and checks if $\mathcal{A}(x, w) = \textbf{True}$. If any w causes $\mathcal{A}$ to accept x, then $\mathcal{A}'(x, w) = \textbf{True}$, and otherwise it outputs **False**. The runtime of $\mathcal{A}'$ is $\exp(\text{poly}(n))$ times the runtime of $\mathcal{A}$, which is still $\exp(\text{poly}(n))$. □

Uncomputability. The next part of the story is that in standard computational models, there are languages $\mathsf{L} \subseteq \{0,1\}^*$ that are undecidable by any algorithm, polytime or not.

To explain this result, we first need to identify algorithms $\mathcal{A}$ with their code. Define a standard way to encode algorithms as binary strings (e.g., write the algorithm in Python and then convert the characters to bits). We let $\alpha \in \{0,1\}^*$ be an arbitrary binary string, and let $\mathcal{A}_\alpha$ denote the algorithm encoded into the binary string α. If α is not an encoding of any algorithm, then we let $\mathcal{A}_\alpha$ be some default algorithm, e.g., the algorithm that ignores its input and immediately outputs **True**. This defines a mapping between algorithms $\mathcal{A}_\alpha$ and binary strings $\alpha \in \{0,1\}^*$. We can view the code α as indexing all of the possible algorithms in the world.

[8] One potential counter is that quantum computing may solve some problems in polytime that classical computing cannot. For this reason, the Church-Turing thesis is sometimes stated with respect to quantum Turing machines.

Now, assume that any algorithm $\mathcal{A}_\alpha$ has a valid behavior on any input $x \in \{0,1\}^*$. For example, the algorithm may ignore all but the first n bits of x, but it can still take x as an input. We can view any algorithm $\mathcal{A}_\alpha$ as partitioning all possible inputs $x \in \{0,1\}^*$ into three sets:

$$x \in \begin{cases} Y(\mathcal{A}) & \mathcal{A}(x) = \textbf{True} \\ N(\mathcal{A}) & \mathcal{A}(x) = \textbf{False} \\ L(\mathcal{A}) & \mathcal{A} \text{ fails to terminate on input } x \end{cases}.$$

In other words, these three sets capture the fact that on any input $x \in \{0,1\}^*$, $\mathcal{A}$ can do one of three things: accept x in finite time, reject x in finite time, or loop indefinitely.

We are now ready to show that there exists a language $\mathsf{L} \subseteq \{0,1\}^*$ that is undecidable by any algorithm (a.k.a. *uncomputable*). The uncomputable language is defined as follows:

$$\mathsf{UC} := \{\alpha \in \{0,1\}^* \mid \alpha \notin Y(\mathcal{A}_\alpha)\}. \tag{4}$$

Said another way, a binary string α is in the language UC iff feeding in the input α to the algorithm $\mathcal{A}_\alpha$ causes the algorithm to either output **False**, or loop indefinitely. The only binary strings α not in UC are those such that $\mathcal{A}_\alpha(\alpha) = \textbf{True}$. We can summarize this behavior as

$$\alpha \in \mathsf{UC} \implies \mathcal{A}_\alpha(\alpha) \neq \textbf{True}, \quad \alpha \notin \mathsf{UC} \implies \mathcal{A}_\alpha(\alpha) = \textbf{True}. \tag{5}$$

Proposition 4. *There is no algorithm that decides* UC.

Proof. Suppose for contradiction that some algorithm $\mathcal{A}_\beta$ decides UC. In other words,

$$\alpha \in \mathsf{UC} \implies \mathcal{A}_\beta(\alpha) = \textbf{True}, \quad \alpha \notin \mathsf{UC} \implies \mathcal{A}_\beta(\alpha) = \textbf{False}.$$

Now consider feeding β as input to $\mathcal{A}_\beta$. If $\beta \in \mathsf{UC}$, then (5) shows that $\mathcal{A}_\beta(\beta) \neq \textbf{True}$, which contradicts the above display. Similarly, if $\beta \notin \mathsf{UC}$, then (5) shows that $\mathcal{A}_\beta(\beta) = \textbf{True}$, which again contradicts the above display. Either way, the algorithm $\mathcal{A}_\beta$ cannot exist. □

The strategy in Proposition 4 (define a sequence, and then define an object that does the opposite of this sequence) is an example of Cantor's *diagonalization argument*. This strategy was originally used to show the existence of uncountable sets (sets with cardinality larger than that of $\mathbb{N}$), and is generally quite useful in establishing impossibility results.

Perhaps the most disappointing thing about Proposition 4 is that the language (4) feels very artificial, as it does not correspond to any human-interpretable problem. This downside is somewhat typical of diagonalization arguments. Luckily, there is a simpler uncomputable language:

$$\mathsf{Halt} := \{(\alpha, x) \in \{0,1\}^* \mid x \notin L(\mathcal{A}_\alpha)\}. \tag{6}$$

In other words, an input of the form (α, x) belongs to Halt iff feeding in the input x to the algorithm $\mathcal{A}_\alpha$ causes it to halt (output either **True** or **False**, but not loop indefinitely).

Corollary 4. *There is no algorithm that decides* Halt.

Proof. Suppose for contradiction that some algorithm $\mathcal{A}$ decides Halt. We show that using $\mathcal{A}$, we can define another algorithm $\mathcal{A}'$ that can decide UC (4), contradicting Proposition 4.

The algorithm $\mathcal{A}'$ is simple. On an input α, we first input (α, α) to $\mathcal{A}$, to check whether $\alpha \in L(\mathcal{A}_\alpha)$. If we determine that $\alpha \in L(\mathcal{A}_\alpha)$, then we can conclude $\alpha \in \mathsf{UC}$ and return **True**. Otherwise, we need to determine whether $\alpha \in Y(\mathcal{A}_\alpha)$ or $\alpha \in N(\mathcal{A}_\alpha)$. We can do so by running $\mathcal{A}_\alpha$ on the input α and waiting for it to terminate, upon which we can determine whether $\alpha \in \mathsf{UC}$. □

Note that Corollary 4 is a reduction from UC to Halt. Conceptually, the message of Corollary 4 is that UC is uncomputable due to inputs that cause algorithms to run forever. If we can rule out this case, then we actually can determine membership in UC. This phenomenon is deep, and in fact there is a complexity class called RE that exactly captures this behavior: languages in RE have a weaker notion of being decided, that allows for infinite looping. It turns out that Halt is a complete language for RE, although we will not cover this in more detail.

Separating NP and NP-hard. For our purposes, the takeaway message of Corollary 4 is that $\mathsf{Halt} \notin \text{TIME}(f(n))$ for any function f. Indeed, $\mathsf{Halt} \in \text{TIME}(f(n))$ implies a finite-time algorithm that decides Halt, contradicting Corollary 4. By Lemma 4, we can conclude $\mathsf{Halt} \notin \text{NP}$.

The next result thus concludes our separation of NP and NP-hard.

Lemma 5. *The language* Halt *is* NP-hard.

Proof. As in Theorem 1, we must show that for all languages $\mathsf{L} \in \text{NP}$, we have $\mathsf{L} \leq_{\text{P}} \mathsf{Halt}$.

How would we decide $x \in \mathsf{L}$ given an access to a Halt oracle? Because $\mathsf{L} \in \text{NP}$, we know that there is a nondeterministic polytime algorithm $\mathcal{A}(x, h)$ that accepts x for some choice of witness h, iff $x \in \mathsf{L}$. Thus, our goal is to simulate the nondeterminism h with the Halt oracle.

Consider an algorithm $\mathcal{A}'$ that behaves as follows: on input x, it computes all of the $\mathcal{A}'(x, h)$ for all of the (finitely many) possible witnesses h. If any of these values evaluate to **True**, then $\mathcal{A}'$ returns **True**. Otherwise, $\mathcal{A}'$ loops forever. Note that even though the runtime of $\mathcal{A}'$ is quite large, we can encode $\mathcal{A}'$ into a succinct description $\alpha = \langle \mathcal{A}' \rangle$, because all $\mathcal{A}'$ needs to know is the code for $\mathcal{A}$, and then it can perform the simulation of computing all $\mathcal{A}(x, h)$ itself.

Now we claim we can decide L with one call to a Halt oracle. Indeed, simply feed (α, x) into the Halt oracle. If Halt returns **True**, this means that $\mathcal{A}'$ halted on x, which means some choice of witness causes $\mathcal{A}$ to accept. Thus we can conclude $x \in \mathsf{L}$ in this case. Otherwise, if Halt returns **False**, then no choice of witness causes $\mathcal{A}$ to accept, so we can correctly conclude $x \notin \mathsf{L}$. □

Time hierarchy theorem. We conclude by mentioning a famous result in complexity theory, that also follows from a diagonalization argument similar to Proposition 4.

Theorem 2 (Time hierarchy theorem)**.** *Let* $f_1 : \mathbb{N} \to \mathbb{N}$ *and* $f_2 : \mathbb{N} \to \mathbb{N}$ *be increasing functions, such that* $f_1(n) = \Omega(n)$ *and* $f_2(n) = \omega(f_1(n) \log f_1(n))$*. There is a language* L *such that* $\mathsf{L} \in \text{TIME}(f_2(n))$*, but* $\mathsf{L} \notin \text{TIME}(f_1(n))$*.*

Theorem 2 says that for any two functions f_1, f_2 serving as time bounds, if f_2 is larger than f_1 by at least a logarithmic factor, then there is a language decidable in time $f_2(n)$, but not in time $f_1(n)$. For example, there is a language decidable in $O(n^3)$ time, but not $O(n^2)$.

It might be tempting to try to use Theorem 2 to prove P ≠ NP, but unfortunately, the languages it constructs are not very explicit: they look at the accept-reject behavior of all algorithms running in $f_1(n)$ time, when fed their own code, and do the opposite. In fact, it remains an open problem to prove that any explicit NP language even requires $\Omega(n \log(n))$ time to decide.

5.2 coNP, NP-intermediate, and the polynomial hierarchy

In this section, we introduce several complexity classes closely related to the P vs. NP conjecture.

coNP. For any language L, define its *complement* $\text{co}(\mathsf{L})$ to be the language that exactly reverses membership with respect to L. That is, for all $x \in \{0,1\}^*$,

$$x \in \mathsf{L} \implies x \notin \text{co}(\mathsf{L}), \quad x \notin \mathsf{L} \implies x \in \text{co}(\mathsf{L}). \tag{7}$$

For example, the complement language $\mathsf{UNSAT} := \text{co}(\mathsf{SAT})$ is defined as follows: an input $x = \langle \Phi \rangle$ is in UNSAT iff Φ corresponds to a CNF formula that is *unsatisfiable*, in that every variable assignment causes Φ to evaluate to **False**. To be consistent with the definition (7), if an input x is not a valid encoding of a CNF formula, then it belongs to UNSAT by default.

Let coNP be the complexity class of all languages $\text{co}(\mathsf{L})$, where $\mathsf{L} \in \text{NP}$. While this is already a fully formal definition, there is a helpful self-contained definition of coNP as consisting of languages where membership is easy to *refute*, that we now provide. Analogously to our definition of NP in Section 3.1, coNP consists of all languages L that are decidable by a nondeterministic polytime algorithm $\mathcal{A}(x, h)$, where $|h| = \text{poly}(|x|)$, with the following properties.

- If $x \in \mathsf{L}$, then for all choices of h, we have that $\mathcal{A}(x, h) =$ **True**.
- If $x \notin \mathsf{L}$, then there is a choice of h such that $\mathcal{A}(x, h) =$ **False**.

Algorithm 3: DecideUNSATcoNP(x, h)

```
Input: x ∈ {0,1}*
if x is an encoding of ⟨Φ⟩ where Φ is a CNF formula then
    if h is an encoding of an assignment to variables in Φ then
        if Φ = True when using the assignment specified by h then
            return False                    // Φ is satisfiable, i.e., x ∉ UNSAT
        end
        return True
    end
end
return True                                 // x or h is not a valid encoding
```

For example, the following nondeterministic algorithm places UNSAT in coNP.

Another example of a natural problem in coNP is the Primes problem. An input $x \in \{0,1\}^*$ belongs to Primes iff $x = \langle p \rangle$ encodes a prime number p. Note that $|x| = O(\log(p))$ (the number of bits needed to describe p), so strategies that try dividing p by all natural numbers up to $\sqrt{p}$, for instance, run in exponential time. However, there is a simple certificate that $x = \langle p \rangle$ does not belong to Primes: we can just provide any factorization $p = q \cdot r$. If we encode the factors $\langle q, r \rangle$ as our hint, a nondeterministic algorithm can easily refute membership in Primes.

We conclude with some helpful facts that help us reason about coNP.

Lemma 6. *If* $\mathsf{A} \leq_P \mathsf{B}$*, then* $\mathsf{A} \leq_P \text{co}(\mathsf{B})$*. Hence, for all languages* L*,* $\mathsf{L} \leq_P \text{co}(\mathsf{L})$*.*

Proof. Recall that a polytime Cook reduction from A to B is just an oracle reduction in the sense of Section 2, where we can call the oracle for B polynomially many times. The first claim follows because given an oracle for co(B), we can reverse all of its answers to recover an oracle for B.

To obtain the second claim, apply the first claim with the fact that $\mathsf{L} \leq_P \mathsf{L}$. □

We next deduce a simple consequence of Lemma 6 about completeness in coNP. As in Section 3.2, we say that a language L is coNP-hard if for all languages $\mathsf{L}' \in$ coNP, we have that $\mathsf{L}' \leq_P \mathsf{L}$. Again, define coNP-complete to be the intersection of coNP and coNP-hard.

Lemma 7. *If* $\mathsf{L} \in$ NP-complete*, then* co(L) $\in$ coNP-complete*.*

Proof. Because $\mathsf{L} \in$ NP, it is clear that co(L) $\in$ coNP, so it remains to show coNP-hardness. Let co(L′) be some language in coNP, such that $\mathsf{L}' \in$ NP. We want to show co(L′) $\leq_P$ co(L). We know that $\mathsf{L}' \leq_P \mathsf{L}$, because $\mathsf{L} \in$ NP-complete. Hence, applying Lemma 6 twice,

$$\text{co}(\mathsf{L}') \leq_P \mathsf{L}' \leq_P \mathsf{L} \leq_P \text{co}(\mathsf{L}).$$

□

Remark 1 (Hardness and reductions)**.** *We have been using the* $\leq_P$ *notation to signify existence of a Cook reduction rather than a Karp reduction (cf. Section 2 for definitions). Let us use the notation* $\mathsf{A} \leq_{P,m} \mathsf{B}$ *to denote that there is specifically a Karp reduction from* A *to* B*.*

The standard definitions of NP-hard *and* coNP-hard *are actually with respect to Karp reductions: language* $\mathsf{L} \in$ NP-hard *iff for all languages* $\mathsf{L}' \in$ NP*, we have that* $\mathsf{L}' \leq_{P,m} \mathsf{L}$*. One can check that all of our* NP-hard *languages in Theorem 1, Propositions 1, 2, 3, and Corollaries 2, 3, 4, all obey this stronger definition. It is a good exercise to check that Lemma 7 also goes through with this definition: a Karp reduction from* L' *to* L *implies a Karp reduction from* co(L′) *to* co(L)*.*

The reason complexity theorists prefer this definition is because it provides closure to NP*: if* $\mathsf{B} \in$ NP *and* $\mathsf{A} \leq_{P,m} \mathsf{B}$*, then* $\mathsf{A} \in$ NP*. The same statement is not known to be true if we instead assume* $\mathsf{A} \leq_P \mathsf{B}$*. For example, many researchers believe that* coNP $\neq$ NP*, but Lemma 6 shows that* co(A) $\leq_P$ A $\leq_P$ B*, so the fact that all* NP *languages reduce to* $\mathsf{B} = \mathsf{SAT}$ *would mean that if* NP *is closed under Cook reductions, then* NP = coNP*. We henceforth ignore these subtleties in this course, but provide this remark for the reader's reference when consulting outside resources.*

NP-intermediate. We next discuss the space "in between" P and NP-complete. Formally, we define NP-intermediate := NP \ NP-complete: the class of all NP languages that are *not* NP-hard.

Our first observation is that if P = NP, then NP-intermediate is not very interesting.

Lemma 8. *If* P = NP*, then* NP-intermediate = $\varnothing$*.*

Proof. We wish to show that if P = NP, there are no languages in NP \ NP-complete; equivalently, that every language $\mathsf{L} \in$ NP is also NP-hard. This means that $\mathsf{L}' \leq_{\mathrm{P}} \mathsf{L}$ for arbitrary $\mathsf{L}' \in$ NP. However, this is true because $\mathsf{L}' \in \mathrm{NP} = \mathrm{P}$ by assumption, so there is a polytime decision algorithm for L' that does not even need use the L oracle. Thus L is (trivially) NP-hard. □

As a converse to Lemma 8, we present a result due to Ladner [Lad75].

Theorem 3 (Ladner's theorem)**.** *If* P ≠ NP*, then* NP-intermediate ≠ $\varnothing$*.*

Ladner's theorem guarantees that if P ≠ NP (a statement that most researchers believe), then there is at least one NP-intermediate language. Unfortunately, Ladner's proof is again based on diagonalization, and does not guarantee any explicit language is NP-intermediate.

Interestingly, under a different assumption, we can say something much more concrete. Specifically, many researchers believe that NP ≠ coNP, which means that there are languages for which membership is easy to certify, but hard to refute. This is a very plausible assumption (how would one refute satisfiability of a CNF in polynomial time?) but it is stronger than P ≠ NP.

Lemma 9. *If* NP ≠ coNP*, then* P ≠ NP*.*

Proof. We prove the contrapositive: if P = NP, then NP = coNP. Indeed, suppose that P = NP. This means that for any language $\mathsf{L} \in$ NP, there is a polytime algorithm $\mathcal{A}$ deciding membership in L. This also means that co(L) is decidable in polytime: we can just reverse the outcome of $\mathcal{A}$. As L was arbitary, all languages in coNP are decidable in polytime, so coNP = P = NP. □

If we are willing to assume that NP ≠ coNP, then we can place specific languages of interest in NP-intermediate, by the following observation (proof deferred to Proposition 9, [Aar16]).

Fact 1. *If* NP ≠ coNP*, then any* $\mathsf{L} \in \mathrm{NP} \cap \mathrm{coNP}$ *also satisfies* $\mathsf{L} \in$ NP-intermediate*.*

The reason that Fact 1 is useful is because we actually know that at least one important language is in the set NP ∩ coNP. To motivate this finding, recall the problem Primes we defined earlier, which we know belongs to coNP. In 1975, Vaughan Pratt found a polytime verifiable certificate for primality [Pra75], showing that in fact, Primes is also in NP. This certificate is rather complicated to describe, but it is based on the existence of *generators* in any prime modulus.

It is tempting to use Fact 1 to conclude that Primes is a likely NP-intermediate problem. This conclusion is not as shocking as it might be otherwise, however, due to a breakthrough originally announced in 2002 by Agrawal, Kayal, and Saxena [AKS04]: Primes ∈ P! Thus, unless it turns out that P = NP, it is immediate from Corollary 1 that Primes is not NP-hard.

Fortunately, there is a close relative of Primes, that to our knowledge is in NP-intermediate \ P: a truly "intermediate" problem. This problem is Factoring, a more fine-grained variant of Primes that accepts any input $x = \langle p, t \rangle$ where $p, t \in \mathbb{N}$, and p has a factor in the range $[2, t]$. For example, $\langle 15, 6 \rangle \in$ Factoring because 5 is a factor of 15. It is straightforward to see that Factoring ∈ NP by providing a small factor of p. In fact, we can also show that Factoring ∈ coNP. To see this, we can refute that an input $\langle p, t \rangle$ is in Factoring by providing a prime factorization of p (efficiently checkable using Primes ∈ P), and then verify that all prime factors of p do not exceed t.

To our knowledge, Factoring is not solvable in polytime by Turing machines, so based on Fact 1, it is a likely candidate for membership in NP-intermediate \ P. Indeed, there are many other natural problems in NP ∩ coNP that we do not have polytime algorithms for, e.g., discrete logarithm, parity games, and certain lattice problems. Our logic equally suggests that these problems may be NP-intermediate. Very interestingly, Factoring is also by far the most famous example of a *quantum advantage*: polytime factoring algorithms are known if one allows use of quantum computing [Sho99]. One takeaway from all of this history is that there seems to be much room for the development of interesting algorithms at the intersection of NP and coNP.

Polynomial hierarchy.

5.3 Beyond time complexity

6 Exponential-time algorithms

6.1 Traveling salesman problem

6.2 SAT

7 Approximation algorithms

7.1 Makespan

7.2 SAT

8 Fine-grained complexity

8.1 3SUM

8.2 APSP

8.3 SETH

Further reading

For more on Sections 2 and 3, see Chapters 34.1 to 34.3, [CLRS22], or Chapters 12.1 to 12.5, [Eri24], or Chapters 8.1 to 8.3, [KT05], or Chapters 19 and 23, [Rou22].

For more on Section 4, see Chapters 34.4 to 34.5, [CLRS22], or Chapters 12.6 to 12.14, [Eri24], or Chapters 8.4 to 8.7, [KT05], or Chapter 22, [Rou22].

Section 5 provides a very brief introduction to advanced topics in complexity theory, which can take an entire semester (or longer) to cover in depth. For the interested reader, we recommend the excellent resources [Sip96, AB09] as significantly more detailed overviews.

For more on Section 6, see Chapter 21, [Rou22].

For more on Section 7, see Chapter 35, [CLRS22], or Chapter J, [Eri24], or Chapter 11, [KT05], or Chapter 20, [Rou22].

Section 8 surveys an active area of complexity theory research, and is largely based on reference material in [Wil18, DIM24], both of which we highly recommend to interested readers.

References

[Aar16] Scott Aaronson. $p \stackrel{?}{=} np$. In *Open Problems in Mathematics*, pages 1–122. Springer, 2016.

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

[AKS04] Manindra Agrawal, Neeraj Kayal, and Nitin Saxena. Primes is in p. *Annals of Mathematics*, 160(2):781–793, 2004.

[CLRS22] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Fourth Edition*. The MIT Press, 2022.

[Coo71] Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the 3rd Annual ACM Symposium on Theory of Computing*, pages 151–158. ACM, 1971.

[Coo88] Stephen A Cook. Short propositional formulas represent nondeterministic computations. *Information Processing Letters*, 26(5):269–270, 1988.

[DIM24] DIMACS Center for Discrete Mathematics and Theoretical Computer Science. Dimacs tutorial on fine-grained complexity. https://www.youtube.com/playlist?list=PLKVCRT3MRed69ZsztrNOFxSBQ3ddToCTv, 2024.

[Eri24] Jeff Erickson. *Algorithms*. 2024.

[Hem19] Lane A. Hemaspaandra. SIGACT news complexity theory column 100. *SIGACT News*, 50(1):35–37, 2019.

[Imp95] Russell Impagliazzo. A personal view of average-case complexity. In *Proceedings of the Tenth Annual Structure in Complexity Theory Conference*, pages 134–147. IEEE Computer Society, 1995.

[Kar72] Richard M. Karp. Reducibility among combinatorial problems. In *Proceedings of a symposium on the Complexity of Computer Computations*, The IBM Research Symposia Series, pages 85–103. Plenum Press, New York, 1972.

[KT05] Jon Kleinberg and Éva Tardos. *Algorithm Design*. 2005.

[Lad75] Richard E. Ladner. On the structure of polynomial time reducibility. *J. ACM*, 22(1):155–171, 1975.

[Lev73] Leonid A Levin. Universal sequential search problems. *Problems of information transmission*, 9(3):265–266, 1973.

[Pra75] Vaughan R. Pratt. Every prime has a succinct certificate. *SIAM J. Comput.*, 4(3):214–220, 1975.

[Rob91] John Michael Robson. An $o(t \log t)$ reduction from ram computations to satisfiability. *Theoretical Computer Science*, 82(1):141–149, 1991.

[Rou22] Tim Roughgarden. *Algorithms Illuminated*. Soundlikeyourself Publishing, 2022.

[Sch78] Claus-Peter Schnorr. Satisfiability is quasilinear complete in nql. *Journal of the ACM (JACM)*, 25(1):136–145, 1978.

[Sho99] Peter W Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM review*, 41(2):303–332, 1999.

[Sip96] Michael Sipser. Introduction to the theory of computation. *ACM Sigact News*, 27(1):27–29, 1996.

[Wil18] Virginia Vassilevska Williams. On some fine-grained questions in algorithms and complexity. In *Proceedings of the international congress of mathematicians*, pages 3447–3487. World Scientific, 2018.