CS 6810: Theory of Computing **Spring 2026**

Lecture 1: January 20, 2026

Lecturer: Mohit Gurumukhani *Scribe: Annabel Baniak*

1 Course Information

Welcome to CS 6810! Please consult the Course Website for information about the Syllabus, content and grading scheme.

Most of the course communication will take place through the EdStem Discussion Board.

As described on the course website, each student will have to act as scribe for at least one lecture throughout the course, counting for 25% of your grade. To sign up for a lecture, please fill out your name by a date in the google sheet.

The recommended textbooks for the course are:

- Computational Complexity: A Modern Approach by Sanjeev Arora and Boaz Barak. You can find a draft of the book here.
- Computational Complexity: A Conceptual Perspective by Oded Goldreich. You can find a draft of the book here.
- Mathematics and Computation by Avi Wigderson. You can find a draft of the book here.

2 Theory of Computing

Let's start with the question, what is computing, and why do we care?

At the most basic level, we're interested in to do tasks and how many resources it takes to do these tasks.

Example: Factoring Integers. A simple task we might be interested in would be to factor integers. Assume we get an input of a 1000 digit non-prime integer, and our goal is to get back the factors of this integer. What are the exact amount of resources required to solve this problem?

The current state of the art way of solving this problem has a time complexity of 10^6 years (a million years) in order to solve this for any random input.

However, this is only the runtime of the best *known* algorithm we currently have. Theoretically, someone could invent a different way of completing the problem tomorrow that solves it in an average of 1 second. As much of the the internet is built using cryptographic security measures relying on the difficulty of this problem, that would obviously demand a major change in technical security.

It would be nice to have some formal bound specifying that, with any implementation of any

algorithm, discovered or not discovered, this problem or any other *must* take a certain amount of time to complete.

In this course, we are interested in these *theoretical* guarantees about the resources a task must use to be completed, rather than evaluating the performance of a particular algorithmic implementation.

Example: Multiplying Matrices Another problem we may be interested in is that of multiplying matrices. Assume as input we receive A, an $n \times n$ matrix, and B, another $n \times n$ matrix. Our goal is to output $A \cdot B$.

There are many different algorithms that can be used to solve this problem. A naive approach results in $O(n^3)$ time complexity. Strassen's algorithm gives a slightly better bound of $O(n^{2.81})$. The current state of the art gives a bound of $O(n^{2.371})$.

By analyzing the problem, we know that the true lower bound for any algorithm solving this problem cannot be less than $O(n^2)$. But is the current state of the art bound the best we can practically hope for, or is there theoretical evidence to suggest a faster algorithm is possible? As matrix multiplication is commonly used in ML programs and can result in a bottleneck or at least very resource intensive step for many ML applications, the question as to whether the current bound can be beat would have significant real-world implications.

Resource Tradeoffs The resources we consider in this course include more than just **time complexity**. Other resources include **space complexity** (space in physical memory), and how much **randomness** an implementation to solve a problem must rely on. When considering various different resources like this, it's important to consider how they compare against each other. This includes asking questions like, can randomness help speed up computation? Are there space-time tradeoffs?

Consider the question: Which is better (has greater computational capacity),

1. An algorithm running in time R with no space restrictions, or
2. an algorithm running in space $R^{0.51}$ with no time restrictions?

A recent argument from Williams '25 proves that algorithm number 2 is more powerful. In the paper, Williams argues that any algorithm which runs in time R (with no space restrictions) can equivalently be run in space $\sqrt{R}$ (with no time restrictions). Thus, the algorithm running with space $R^{0.51}$ with no time restrictions allows for strictly greater computational complexity than the algorithm running with time R with no space restrictions.

Formalization of Computation To analyze questions like this, we want to formalize our notion of computation, and what we mean by a computational task.

Let's represent any computational task by a function f:$\{0,1\}^* \to \{0,1\}$ over the binary alphabet, giving a yes/no answer. Equivalently, we can represent $f : \Sigma^* \to \Sigma^*$ over any finite alphabet.

For any such function f, we want a theoretical model of a computer we can reason about to make formal guarantees about f. To do this, we use...

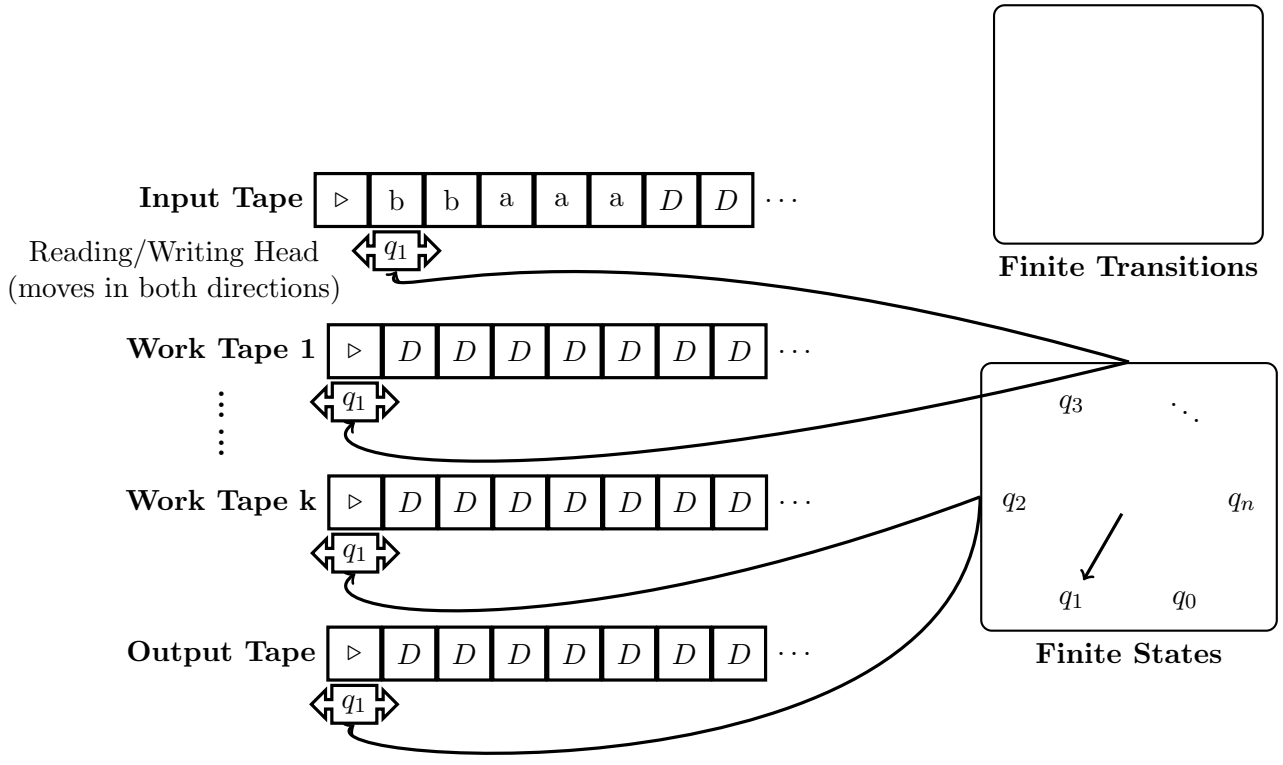
3 Turing Machines

Turing Machines (1936) are pencil-and-paper models that capture natural computation and are used to calculate these formal guarantees. Independently, Church and Godel also made natural models of computation around the same time, but they're equivalent to Turing's model:

Church-Turing Hypothesis. Any other powerful natural model of computation is equivalent to a Turing machine.

Intuition. The idea behind a Turing Machine is a formal model of how a human solves a problem: You work with a potentially infinite amount of paper/scratchpad in front of you, and proceed working through the problem using finite set of rules in your brain.

Similarly, a Turing Machine is made up of tapes, representing the infinite amount of scratchpad space, and tape heads which move along the tapes according to a finite set of rules. Our model of a TM takes an input tape with the input string, and maintains k worktapes as well as an output tape. A finite state module maintains the current position of the tape heads on the tapes, and a finite transition model records the finite set of rules used to move the tape heads:



1

[1] Turing Machine LaTeX diagrams adapted from Sebastian Sardina.

Formal Definition of a TM. We define a TM (Turing Machine) formally as the tuple $(Q, \Sigma, \Gamma, \delta)$, where:

- Q is a state set, where a state represents some information about the current position of the machine, for example a 'start state' or 'copying state'. Q must contain at least one 'halt state' wherein the machine stops its process.
- Σ is the Input Alphabet and Output Alphabet
- Γ is the Tape Alphabet, where $\Sigma \subseteq \Gamma$, and there exist the symbols $D \in \Gamma$ representing blank positions on the tape and $\triangleright \in \Gamma$ representing the start position of the tapes, as they are only infinite in one direction.
- δ is the transition function taking one state of the machine and moves the tape heads to form a new state, where $\delta : Q \times \Sigma^2 \times \Gamma^k \to Q \times \Sigma \times \Gamma^k \times \{L, S, R\}^{k+2}$

The type signature for δ specifies that at each step, the transition function takes in the current state and the characters currently read from the 2 input/output tapes, as well as the characters read from the k worktapes. As output, it gives a new state, generated by first overwriting the current position on the output tape with an element of Σ, and overwriting the current position on the k worktapes with elements from Γ, and then moving each of the $(k+2)$ tape heads in one of three directions: move Left, move Right, or Stay the same.

Now we can reason about the time a TM takes to process a computing problem, by counting how many steps the machine takes to reach a halt state.

Finite Encodings. Note that this formal definition gives a **finite encoding** of any Turing Machine, where each element of the tuple is represented using a finite binary encoding. So, **any TM can be described by a finite binary string**.

Similarly, the opposite direction also holds; **any finite binary string can be associated to a TM which represents it**. If the string follows the correct way of encoding a TM, let it encode that machine, and otherwise, let the string be associated to the TM which always halts and outputs 0.

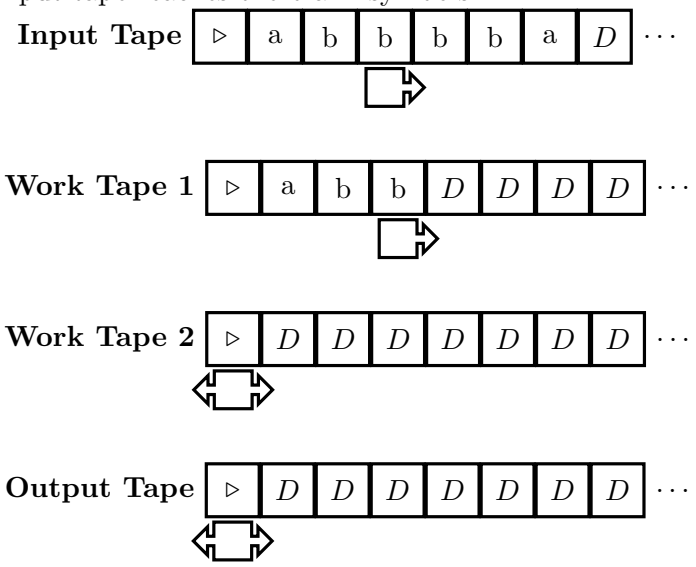
This allows TMs the capacity to encode anything that can be encoded by a finite binary string, for example any program written in a high-level programming language like python. All programs written in high-level programming languages can be compiled into binary to get a finite file, which can then be encoded into a Turing Machine.

4 Example 1 - Palindromes

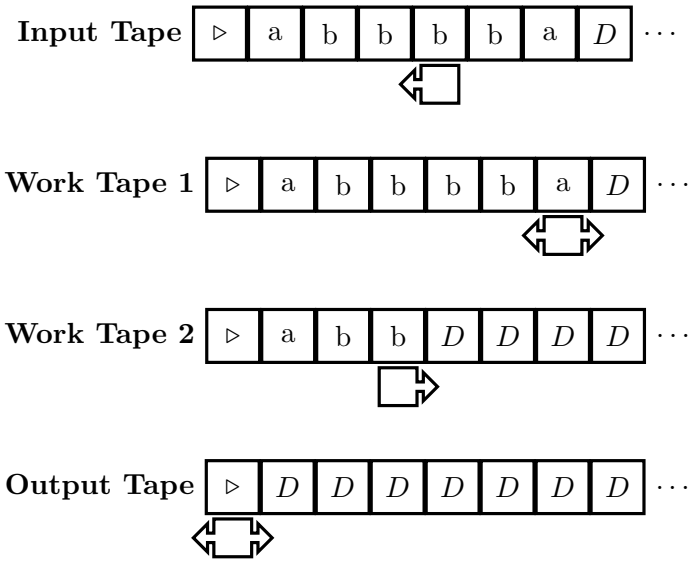
Consider the function $f : \{0,1\}^* \to \{0,1\}$ which takes a binary string $x \in \{0,1\}^*$, and returns 1 if x is a palindrome and 0 otherwise. For example, input word $f(0110)$ would return 1 but $f(0111)$ would return 0.

We can model this computational task using a Turing Machine. Create a TM M with 2 worktapes, and given the input string x as the contents of the input tape. Then M can perform the task using the following:

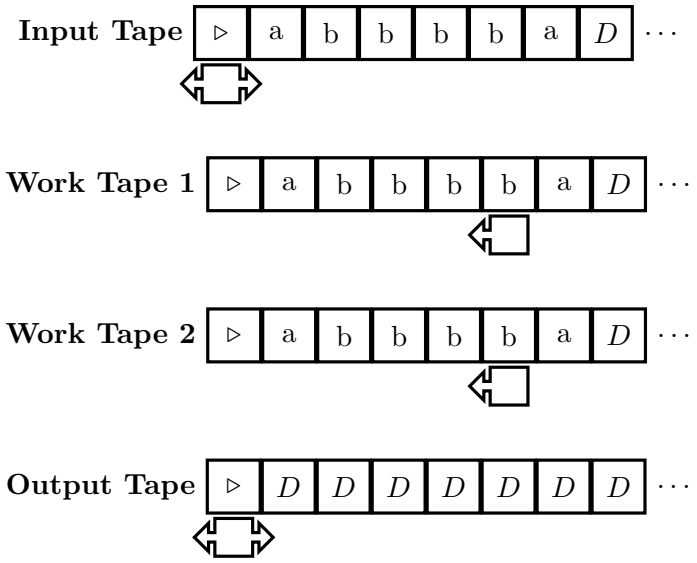
1. Copy over the entire input tape to the first worktape, stopping when the tape head on the input tape reaches the blank symbols.



2. Copy over the entire input tape to the second worktape, but reversed. As from step 1, the input tape head was left at the first blank symbol, it is in the perfect position to copy over the contents of the input tape backwards.



3. Compare the contents of worktape 1 and worktape 2, moving from right to left (as both tape heads on both worktapes are now at the end of the non-blank symbols in their word) until you reach the start symbol ▷. If you reach ▷ without any elements of the worktapes disagreeing, then put 1 into the output tape, else put 0.



This process occurs with $O(n + n + n) = O(n)$ steps. However, if we were restricted to a TM with only 1 tape, where the single tape operates as input tape, worktape, and output tape, this process would take $O(n^2)$ time. But it would still be *possible* if we only had a TM with one tape.

5 Multi-tape and Single Tape Equivalence

More worktapes do not guarantee more power in Turing Machines. In fact, every action that can be performed with a multi-tape TM can also be performed with a single tape TM.

Theorem 1. $\forall k$-tape TMs M, $\exists$ a TM M' with only one worktape such that M and M' compute the same function.

However, while the computational complexity between M and M' are equal, the time they take to execute a computational task is not. If M takes time $k \times T(n)$ time to execute a task, then M' will take $k \times (T(n))^2$ time to execute the task.

Proof. Given a TM M with k worktapes, we will create an M' with only one worktape.

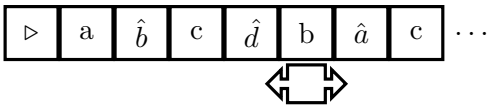
Let the worktape of M' be split into sections representing the k worktapes of M, to infinity, so that the ith position on the worktape of M' represents the (i mod k)th worktape of M. The character filling this ith position of the worktape of M' represents the current character being read from the (i mod k)th worktape of M.

However, as there are thus infinitely representations of each worktape of M present in M', we need a way of signifying which is the 'current' representation of the tapes of M. So, let Γ', the tape alphabet of M', be equal to $\Gamma' = \{a, \hat{a} \mid a \in \Gamma\}$, for Γ the tape alphabet from M.

Now, the one 'hat character' present in the worktape of M' for each index class mod k represents the true current character at the head of the i mod kth worktape from M.

For example, if we were translating a TM M with three worktapes, our worktape in M' may look like:

M' **worktape**



Here, assuming the start symbol ▷ is at index 0, the indices 1, 4, 7, etc represent the first worktape from M, but only index 4 represents the current character at the head of worktape 1 - d, as it is the only 'hat' character.

To simulate each step of M in M',

- We scan the worktape of M' to find these 'hat' characters to extract the current information of M, and store this as M's state.
- Then, we can use the transition function δ from M to calculate the next moves to perform in M'
- Finally, we apply these changes to the worktape of M' from right to left.

Note that one step of M took $O(n)$ steps in M'. So if it took $T(n)$ total steps to run the program on M, it takes $(T(n))^2$ steps to run it on M'.

Theorem 2. Instead of using an arbitrary tape alphabet Γ, we can simulate any TM with the minimal alphabet $\Gamma = \{0, 1, D, \triangleright\}$.

We leave the proof as exercise for the reader, as after representing the elements of Γ in binary encoding, the process is similar as the process for Theorem 1.

Together, Theorems 1 and 2 support the Church-Turing hypothesis, which argues that any machine modeling computability can be converted into a Turing Machine.

6 Universal Turing Machine

Recalling the definition of encoded TMs, we can now discuss the **Universal Turing Machine**, U. This TM takes as input the binary encoding of some TM, (M), and a binary string x. U then outputs M(x) (M run with x as input).

The real power of the Universal TM is that U has finite description but can run any TM on any string.

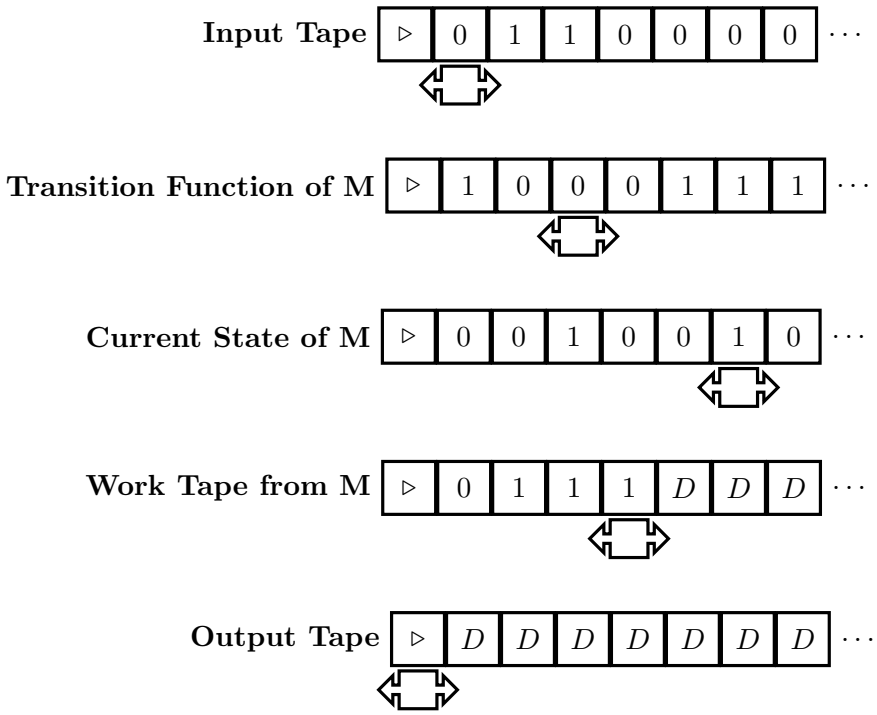
Implementing the Universal TM. Given input strings (M) and x, first transform M to have 1 worktape and tape alphabet $\{0, 1, D, \triangleright\}$, using the processes discussed in the above section.

Now given this, we can assume that M operates with one worktape and only with tape alphabet $\{0, 1, D, \triangleright\}$. Let U have an input tape, three worktapes, and an output tape.

- On worktape one, copy over the binary encoding of the transition function of M.
- Worktape two will work to simulate the current state of M in U.

- Worktape three will serve as the single worktape from M.

Now we can simulate M one step at a time, using the rules from the transition function provided in worktape 1 to update worktapes 2 and 3 (the state and worktape of M) with each step.



Note that the number of tapes, the tape alphabet, and the number of states of U is independent of those qualities in M.

If M takes time $T(n)$, then U should take time $T(n)^2$ (naive) to simulate it, but $T(n)log(T(n))$ time with a 'clever' implementation.

7 Can TMs compute all functions?

All this about the power of Turing Machines begs the question: Are TMs Computationally complex enough to compute all possible functions? Given an arbitrary $f : \{0,1\}^* \to \{0,1\}$, does there exist an TM M which computes f?

The answer is unfortunately no.

Theorem 3. TMs cannot compute all possible functions.

Note that the number of functions is uncountable; there are $2^{\mathbb{N}}$ functions of the type $\{0,1\}^* \to \{0,1\}$. However, we showed earlier that there is a bijection between the set of TMs and the set of finite binary strings, which is a countable set, of size $\mathbb{N}$. By Cantor's diagonalization proof, there cannot exist a bijection between all functions and all TMs, and there must be some functions that cannot be captured by a TM.

U.C. To illustrate this, we can define a new function, U.C. which represents this diagonalizational argument. Let U.C. take as input the binary string α.

- We know α must associate to some TM, M_α, so let U.C. call M_α on input string α.
- If this call halts in a finite number of steps, and $M_\alpha(\alpha) = 1$, then let U.C. output 0.
- Otherwise, let U.C. output 1.

Now we can claim that **No TM can ever compute U.C.**.

Proof. Suppose such a TM $\tilde{M}$ existed that exactly computes U.C..

Let $x = (\tilde{M})$ be the binary string encoding of $\tilde{M}$, and consider $\tilde{M}(x)$, or in other words, the encoding of $\tilde{M}$ fed in as the input string to $\tilde{M}$ itself.

- If $\tilde{M}(x) = 1$, then $\tilde{M}(\ (\tilde{M})\ ) = 1$. Consider U.C.$(x)$. By the definition of U.C., U.C. takes the TM associated with input string x, which in this case is exactly $\tilde{M}$, and if $\tilde{M}(x) = 1$, U.C. returns 0. So U.C.$(x) = 0$.
- If $\tilde{M}(x) = 0$, then $\tilde{M}(\ (\tilde{M})\ ) = 0$. Consider U.C.$(x)$. By the definition of U.C., U.C. takes the TM associated with input string x, which in this case is exactly $\tilde{M}$, and if $\tilde{M}(x) = 0$, then U.C. is in the "otherwise" case and returns 1. So U.C.$(x) = 1$.

Thus, in both cases, this presents a contradiction, as we assumed at the start that $\tilde{M}$ exactly computed U.C., but they in fact always differ on the input string $x = (\tilde{M})$.

Therefore, no TM exists that can encode U.C.

However, this is a bit of a strangely constructed function. It would be nice to have a function which cannot be modeled by a TM with a more natural construction...

8 Halting Problem

...which we find in the Halting Problem.

Halting Problem Let's define the function HALT as follows: For any input binary strings α, x, HALT$(\alpha, x) = 1$ iff M_α(x) halts in finite time, where M_α is the TM encoded by the string α.

Theorem 4. HALT is incomputable, and cannot be represented by any TM.

Proof. Suppose for contradiction that a TM M_H computes HALT.

Then we can construct a TM $M_{U.C.}$ which takes in an arbitrary binary string $\alpha \in \{0,1\}^*$, and define $M_{U.C.}$ such that:

- $M_{U.C.}$ calls U, the Universal TM, to simulate M_H on the input string α.
- If this simulation does not halt, then $M_H(\alpha) = 0$, so let $M_{U.C.}$ output 1.

- Otherwise, if the simulation of M_H on α does halt, we let $M_{U.C.}$ output the negation of the universal call, $NOT\ U(M_\alpha, \alpha)$.

Now, notice that $M_{U.C.}$ takes in an arbitrary binary string $\alpha \in \{0,1\}^*$ and outputs the exact definition of *U.C.* from above. Thus, $M_{U.C.}$ exactly computes *U.C.*.

However, we have already showed that this isn't possible, so we reach a contradiction.

We have reduced the U.C. problem to the Halting problem and thus have shown that if we could compute HALT, then we could compute U.C.. By contrapositive, we know that since we can't compute U.C., then we can't compute HALT.

This proof structure is called a **reduction**. When reducing algorithm B to algorithm A, if we can solve A, we can solve B. Therefore, by contrapositive, if we know B is not solvable by our current knowledge, then we know A is not solvable by our current knowledge either. If we encounter a new problem A in the wild, we can reduce to it from a problem we already know we cannot solve, B, to show that A is not solvable either.