Notes on Complexity Theory

Richard Willie

November 10, 2024

Contents

1 Diagonalization

A basic goal of complexity theory is to prove that certain complexity classes (e.g. **P** and **NP**) are not the same. To do so, we need to exhibit a machine in one class that differs from every machine in other class in the sense that their answers are different on at least one input. This chapter describes **diagonalization**—essentially the only general technique known for constructing such a machine.

In this chapter, we will use diagonalization in clever ways. We first use it in Section 1.1 and Section 1.2 to prove hierarchy theorems, which show that giving Turing machines more computational resources allows them to solve a strictly larger number of problems. We then use diagonalization in Section 1.3 to show a fascinating theorem of Ladner: If $\mathbf{P} \neq \mathbf{NP}$, then there exists a problem that are neither **NP**-complete nor in **P**.

Though diagonalization led to some of these early successes of complexity theory, researchers concluded in the 1970s that diagonalization alone may not resolve **P** vs **NP** and other interesting questions. Section 1.4 describes their reasoning. Ironically, these limits of diagonalization are proved using diagonalization itself.

§1.1 Time Hierarchy Theorem

Common sense suggests that giving a Turing machine more time should increase the class of problems that it can solve. For example, Turing machines should be able to decide more languages in time n^3 than they can in n^2. The **time hierarchy theorem** proves that this intuition is correct, subject to certain conditions described below. We use the term "hierarchy theorem" because this theorem proves that time complexity classes aren't all the same—they form a hierarchy whereby the classes with larger bounds contain more languages than the classes with smaller bounds.

We begin with the following technical definition, given by Goldreich [Go08].

Definition 1.1.1 (Time-constructible functions)

A function $f : \mathbb{N} \to \mathbb{N}$ is called **time-constructible** if there exists a TM M which, given a string $\mathtt{1}^n$, outputs the binary representation of $f(n)$ in $O(f(n))$ time.

Example 1.1.1

All the commonly used functions $f(n)$ are time-constructible, as long as $f(n)$ is at least $c \cdot n$ for some constant $c > 0$. No function which is $o(n)$ can be time-constructible unless it is eventually constant, since there is insufficient time to read the entire input.

When designing algorithms of arbitrary time complexity $f : \mathbb{N} \to \mathbb{N}$, we need to make sure that the algorithm does not exceed the time bound. However, when invoked on an input w, we cannot assume that the algorithm is given the time bound $f(|w|)$ explicitly.

A reasonable design methodology is to have the algorithm compute this bound before doing anything else. This, in turn, requires the algorithm to read the entire input and compute $f(n)$ in $O(f(n))$ steps; otherwise, this preliminary stage already consumes too much time. The latter requirement motivates the notion of time-constructible functions.

There are multiple definitions of the time hierarchy theorem. We present the following one by Sipser [Si13], with some correction.

Theorem 1.1.2 (Time hierarchy theorem)

For any time-constructible function $f : \mathbb{N} \to \mathbb{N}$ where $f(n) = \omega(n \log n)$, a language A exists that is decidable in $O(f(n))$ time but not decidable in time $o(f(n)/\log f(n))$.

We present the following proof by Sipser [Si13], with some modification.

Proof Idea. We must demonstrate a language A that has two properties. The first says that A is decidable in $O(f(n))$ time. The second says that A isn't decidable in $o(f(n)/\log f(n))$ time.

We describe A by giving an algorithm D that decides it. Algorithm D runs in $O(f(n))$ time, thereby ensuring the first property. Furthermore, D guarantees that A is different from any language that is decidable in $o(f(n)/\log f(n))$ time, thereby ensuring the second property. Notice that language A is different from other languages in that it lacks a non-algorithmic definition. Therefore, we cannot offer a simple mental picture of A.

In order to ensure that A is not decidable in $o(f(n)/\log f(n))$ time, we design D to implement the diagonalization method. If M is a TM that decides a language in $o(f(n)/\log f(n))$ time, D guarantees that A differs from M's language in at least one place. Which place? The place corresponding to a description of M itself.

Let's look at the way D operates. Roughly speaking, D takes its input to be the description of a TM M. (If the input isn't the description of any TM, then D's action is inconsequential on this input, so we arbitrarily make D reject.) Then, D runs M on the same input—namely, $\langle M \rangle$—within the time bound $\lceil f(n)/\log f(n) \rceil$. If M halts within that much time, D accepts iff M rejects. If M doesn't halt, D just rejects. So if M runs within time $\lceil f(n)/\log f(n) \rceil$, D has enough time to ensure that its language is different from M's. If not, D doesn't have enough time to figure out what M does, but fortunately D has no requirement to act differently from machines that don't run in $o(f(n)/\log f(n))$ time, so D's action on this input is inconsequential.

This description captures the essence of the proof but omits several important details. If M runs in $o(f(n)/\log f(n))$ time, D must guarantee that its language is different from M's language. But even when M runs in $o(f(n)/\log f(n))$ time, it may use more than $f(n)$ time for small n, when the asymptotic behavior hasn't "kicked in" yet. Possibly, D might not have enough time to run M to completion on the input $\langle M \rangle$, and hence D will miss its opportunity to avoid M's language. So, if we aren't careful, D might end up deciding the same language M decides, and the theorem wouldn't be proved.

We can fix this problem by modifying D to give it additional opportunities to avoid M's language. Instead of running M only when D receives the input $\langle M \rangle$, it runs M whenever it receives an input of the form $\langle M \rangle \, \mathtt{10}^*$, that is, an input of the form $\langle M \rangle$ followed by a `1` and some number of `0`s. Observe that by doing this, we get to simulate M on infinitely many w, so now D has infinitely many opportunities to avoid M's language. Then, if M

really is running in $o(f(n)/\log f(n))$, we will eventually encounter a sufficiently large w, where $w = \langle M \rangle \, \mathtt{1}\mathtt{0}^k$ for some large value of k, and D will have enough time to run it to completion because the asymptotic must eventually kick in.

One last technical point. The simulation of M by D introduces a logarithmic factor overhead. This overhead is the reason for the appearance of the $1/\log f(n)$ factor in the statement of this theorem. If we had a way of simulating a single-tape TM by another single-tape TM for a prespecified number of steps, using only a constant factor overhead in time, we would be able to strengthen this theorem by changing $o(f(n)/\log f(n))$ to $o(f(n))$. No such efficient simulation is known. □

We now describe the proof formally.

Proof. The following $O(f(n))$ time algorithm D decides a language A that is not decidable in $o(f(n)/\log f(n))$ time.

D = On input w:

1. If w is not of the form $\langle M \rangle \, \mathtt{1}\mathtt{0}^*$ for some TM M, *reject*.
2. Let n be the length of w.
3. Compute $f(n)$ using time constructibility, and store the value $\lceil f(n)/\log f(n) \rceil$ in a binary counter. Decrement this counter before each step used to carry out stage 4. If the counter ever hits 0, *reject*.
4. Simulate M on w.
5. If M accepts, then *reject*. If M rejects, then *accept*.

We examine each of the stages of this algorithm to determine the running time. Obviously, stages 1, 2, and 3 can be performed within $O(f(n))$ time. In stage 4, every time D simulates one step of M, it takes M's current state together with the tape symbol under M's tape head and looks up M's next action in its transition function so that it can update M's tape appropriately. All three of these objects (state, tape symbol, and transition function) are stored on D's tape somewhere. If they are stored far from each other, D will need many steps to gather this information each time it simulates one of M's steps. Instead, D always keep this information close together.

We can think of D's single tape as organized into *tracks*. One way to get two tracks is by storing one track in the odd positions and the other in the even positions. Alternatively, the two-track effect may be obtained by enlarging D's tape alphabet to include each pair of symbols, one from the top track and the second from the bottom track. We can get the effect of additional tracks similarly. Note that multiple tracks introduce only a constant factor overhead in time, provided that only a fixed number of tracks are used. Here, D has three tracks.

One of the tracks contains the information on M's tape, and a second contains its current state and a copy of M's transition function. During the simulation, D keeps the information on the second track near the current position of M's head on the first track. Every time M's head position moves, D shifts all the information on the second track to keep it near the head. Because the size of the information on the second track depends only on M and not on the length of the input to M, the shifting adds only a constant factor to the simulation time. Furthermore, because the required information is kept close together, the cost of looking up M's next action in its transition function and

updating its tape is only a constant. Hence, if M runs in $g(n)$ time, D can simulate it in $O(g(n))$ time.

At every step in stage 4, D must decrement the step counter originally set in stage 3. Here, D can do so without adding excessively to the simulation time by keeping the counter in binary on a third track and moving it to keep it near the present head position. This counter has a magnitude of about $f(n)/\log f(n)$, so its length is $\log(f(n)/\log f(n))$, which is $O(\log f(n))$. Hence, the cost of updating and moving at each step adds a $\log f(n)$ factor to the simulation time, thus brining the total running time to $O(f(n))$. Therefore, A is decidable in time $O(f(n))$.

Now, we show that A is not decidable in $o(f(n)/\log f(n))$ time. Assume the contrary that some TM M decides A in time $g(n)$, where $g(n)$ is $o(f(n)/\log f(n))$. Here, D can simulate M, using time $d \cdot g(n)$ for some constant d. If the total simulation time (not counting the time to update the step counter) is at most $f(n)/\log f(n)$, the simulation will run to completion. Because $g(n)$ is $o(f(n)/\log f(n))$, some constant n_0 exists where $d \cdot g(n) < f(n)/\log f(n)$ for all $n \geq n_0$. Therefore, D's simulation of M will run to completion as long as the input has length n_0 or more. Consider what happens when we run D on the input $\langle M \rangle \, \mathtt{1}\mathtt{0}^{n_0}$. This input is longer than n_0 so the simulation in stage 4 will complete. Therefore, D will do the opposite of M on the same input. Hence, M doesn't decide A, which contradicts our assumption. Therefore, A is not decidable in $o(f(n)/\log f(n))$ time. □

Remark 1.1.3 — Actually, there is one other technical detail that we conveniently ignored in our proof, in order to keep it concise. Let us discuss the detail here. In Stage 4 of D, it simulates M on w. This simulation may require representing each cell of M's tape with several cells on D's tape because M's tape alphabet is arbitrary while D's tape alphabet is fixed. However, initializing the simulation by converting D's input w to this representation involves rewriting w so that its symbol are spread apart by several cells. Let n be the length of w. If we use the obvious copying procedure for spreading w, this convention would involve $O(n^2)$ time and that would exceed the $O(f(n))$ time bound for small f. Instead, we observe that D operates on an input w of the form $\langle M \rangle \, \mathtt{1}\mathtt{0}^k$, and we only need to carry out the simulation when k is sufficiently large. We consider only $k > |\langle M \rangle|^2$. We can spread out w by first using the obvious copying procedure for $\langle M \rangle$ and then counting the trailing 0s and rewriting these by using that count. The time for spreading $\langle M \rangle$ is $O(|\langle M \rangle|^2)$ which is $O(n)$. The time for counting the 0s is $O(n \log n)$, which is $O(f(n))$.

Remark 1.1.4 — We also didn't elaborate on the fact that our function $f(n)$ is $\omega(n \log n)$ in the statement of our theorem. This is because, as we will demonstrate in Corollary 1.1.6, the time hierarchy theorem can be used to distinguish between two complexity classes, e.g. $\mathbf{DTIME}(f(n))$ and $\mathbf{DTIME}(g(n))$. We require both functions f and g to be time-constructible.

Remark 1.1.5 — Hartmanis and Stearns [HaSt65] were the first to establish the time hierarchy theorem. A year later, Hennie and Stearns [HeSt66] improved this result by demonstrating that t steps on any k-tape TM can be simulated in $O(t \log t)$ steps on a two-tape TM. Although we managed to obtain an equally sharp time

hierarchy, it's worth noting that we proved our theorem by simulating a single-tape TM with another single-tape TM. This specific approach was originally established by Hartmanis and Hopcroft [Ha68]. Curiously, their original proof did not rely solely on simulation; it distinguished between two cases:

1. When the running time is $\Omega(n^2)$, simulation is used, much like what we did in our proof.
2. When the running time is $o(n^2)$, they presented a language directly for the separation, without using simulation. This part of the proof depended on specific properties of single-tape TMs with sub-quadratic running times.

Verify this. Also clarify whose running time? The machine simulating or simulated?

Now we can introduce the following important corollary, which is very frequently referred to as the time hierarchy theorem in other literature.

Corollary 1.1.6

For any two time-constructible functions $f, g : \mathbb{N} \to \mathbb{N}$ where $f(n) \log f(n) = o(g(n))$,

$$\mathbf{DTIME}(f(n)) \subsetneq \mathbf{DTIME}(g(n)).$$

This corollary allows us to separate various deterministic time complexity classes. For example, we can show that the function n^c is time-constructible for any natural number c. Hence, for any two natural numbers $c_1 < c_2$ we can prove that $\mathbf{DTIME}(n^{c_1}) \subsetneq \mathbf{DTIME}(n^{c_2})$. With a bit more work we can show that n^c is time-constructible for any rational number $c > 1$ and thereby extend the preceding containment to hold for any rational numbers $1 \leq c_1 < c_2$. Observing that two rational numbers c_1 and c_2 always exist between any two real numbers $\epsilon_1 < \epsilon_2$ such that $\epsilon_1 < c_1 < c_2 < \epsilon_2$ we obtain the following additional corollary demonstrating a fine hierarchy within the class $\mathbf{P}$.

Corollary 1.1.7

For any two real numbers $1 \leq \epsilon_1 < \epsilon_2$,

$$\mathbf{DTIME}(n^{\epsilon_1}) \subsetneq \mathbf{DTIME}(n^{\epsilon_2}).$$

We can also use the time hierarchy theorem to separate the classes $\mathbf{P}$ and $\mathbf{EXPTIME}$.

Corollary 1.1.8

$\mathbf{P} \subsetneq \mathbf{EXPTIME}$.

Proof. We have

$$\mathbf{DTIME}(2^n) \subseteq \mathbf{DTIME}\left(o\left(\frac{2^{2n}}{2n}\right)\right) \subsetneq \mathbf{DTIME}(2^{2n})$$

by the time hierarchy theorem. It follows that

$$\mathbf{P} \subseteq \mathbf{DTIME}(2^n) \subsetneq \mathbf{DTIME}(2^{2n}) \subseteq \mathbf{EXPTIME}.$$

□

This corollary establishes the existence of decidable problems that are decidable in principle but not in practice—that is, problems that are decidable but intractable.

Remark 1.1.9 — In fact, we can show that **P** is strictly contained within **EXPTIME** without explicitly relying on the time hierarchy theorem. Instead, we'll provide a direct proof using—you guessed it—diagonalization!

Proof. Let

$$L = \{(\langle M \rangle, x, 1^n) \mid M(x) \text{ halts in at most } 2^n \text{ steps}\}.$$

Clearly $L \in \mathbf{EXPTIME}$. On the other hand, suppose by contradiction that $L \in \mathbf{P}$. Then there's some polynomial-time Turing machine A such that $A(w)$ accepts if and only if $w \in L$. Let A runs in $p(n + |\langle M \rangle| + |x|)$ on input $(\langle M \rangle, x, 1^n)$, for some polynomial function p. Then using A, we can easily produce another machine B that does the following.

> B = On input $w = (\langle M \rangle, 1^n)$:
>
> 1. Runs forever if $M(\langle M \rangle, 1^n)$ halts in at most 2^n steps; otherwise halts.

Note that, if B halts at all, then it halts after only $p(2n + 2\langle M \rangle) = n^{O(1)}$ steps. Now consider what happens when B is run on input $(\langle B \rangle, 1^n)$. If $B(\langle B \rangle, 1^n)$ runs forever, then $B(\langle B \rangle, 1^n)$ halts. Conversely, if $B(\langle B \rangle, 1^n)$ halts, then for all sufficiently large n, it halts in fewer than 2^n steps, but that means that $B(\langle B \rangle, 1^n)$ runs forever. So we conclude that B, and hence A, cannot exist. □

§1.2 Nondeterministic Time Hierarchy Theorem

The time hierarchy theorem for nondeterministic Turing machines was originally proven by Cook [Co72]. The following tighter result is due to Seiferas, Fischer, and Meyer [SeFiMe78].

Theorem 1.2.1 (Nondeterministic time hierarchy theorem)
For any two time-constructible functions $f, g : \mathbb{N} \to \mathbb{N}$ where $f(n+1) = o(g(n))$,

$$\mathbf{NTIME}(f(n)) \subsetneq \mathbf{NTIME}(g(n)).$$

Remark 1.2.2 — The effect of the term "+1" in $f(n+1)$ is pretty significant. Specifically, it makes a difference to all time-constructible function $t : \mathbb{N} \to \mathbb{N}$ where $t(n) = \omega(2^n)$. For instance, consider the functions $f_1(n) = 2^{n^2}$ and $f_2(n) = 2^{(n+1)^2}$. Our hierarchy theorem is not sharp enough to separate the classes $\mathbf{NTIME}(f_1(n))$ and $\mathbf{NTIME}(f_2(n))$, because $f_1(n+1) \neq o(f_2(n))$.

Our first instinct is to duplicate the proof of Theorem 1.1.2, since there is a universal TM for nondeterministic computation as well. (In fact, the simulation of an NTM by another NTM is very efficient, incurring only a constant factor overhead in time [ArBa09].) The

problem, however, is that we don't know how such a construction would work on our new machine D. In particular, it remains unclear how we could "flip the answer" of M—to *efficiently* compute, given the description of an NTM M and an input w, the value $1 - M(w)$. We give the following hypothetical (erroneous) construction of D to illustrate our point.

D = On input w:

1. Let n be the length of w.
2. If w is not of the form $\langle M \rangle \, \mathtt{10}^*$ for some TM M, *reject*.
3. Nondeterministically simulate M on w for up to $g(n)$ steps.
   - 3.1. If M accepts, then *reject*.
   - 3.2. If M rejects, then *accept*. (How to decide this efficiently so the simulation does not timeout?)

A nondeterministic Turing machine accepts if at least one path accepts; it rejects only when *all paths reject*. This asymmetry makes it hard to flip answers efficiently. For instance, suppose we have an NTM M that has two paths for input w where one accepts and the other rejects. M has at least one accepting path for w, so it accepts. Suppose we want to produce a machine that accepts exactly the input that M rejects. The obvious attempt is to take M and make its accepting state reject, and its rejecting states accept. However, this new machine M' has one rejecting path and one accepting path. So it still accepts w, which it was supposed to reject.

One limitation of the nondeterministic Turing machine is that it can't look at all its paths simultaneously and take action based on what all of those paths do. When simulating an NTM M with another machine M', each path M' can only simulate one path of M and has no knowledge of the other paths. It cannot decide to accept/reject based on the outcomes of paths it cannot see. Therefore, each path of M' can only follow simple rules based on its own outcome. When M' discovers that M rejects the input w on a certain path, M' still *cannot* verify whether w is in the language of M or not (because of the asymmetric accept/reject condition of the NTM). Therefore, it seems that the only reliable way to determine if M rejects w is to exhaustively check every possible path, which is exponentially more time-consuming than checking a single path.

What now? There are two options here:

1. We admit defeat. Accept the exponential-time slowdown, but then we won't get a fine hierarchy at all.
2. Or, we try to come up with a better solution. Consider the following idea. Now, our new simulator is in no hurry to diagonalize, it will not try to flip the answer of a subroutine NTM on every input, but only on a crucial input out of a sufficiently large interval. This technique is known as **lazy diagonalization**. We show that it does suffice to establish our hierarchy theorem.

We present the following proof, inspired by Žák [**Žá83**].

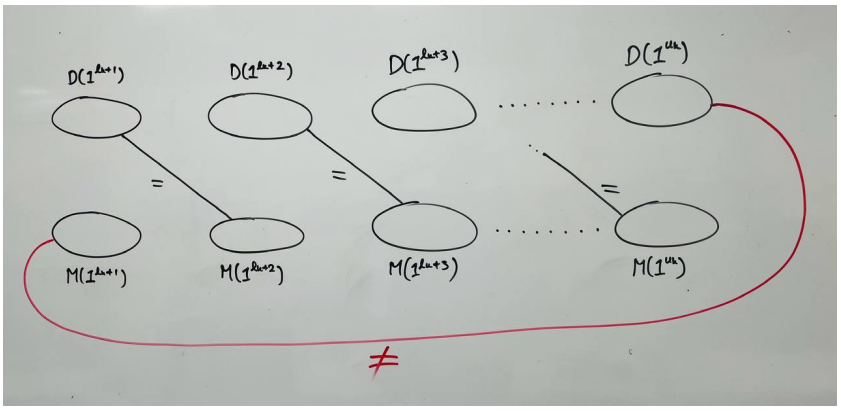
Proof Idea. Recall that in our proof of the deterministic time hierarchy theorem, D gets to simulate M infinitely often. When the input strings get long enough, D accepts iff M rejects. This means that D exhibits different behavior from M on infinitely many inputs. This diagonalization technique demonstrates that no other TM M (that is significantly more efficient than D) can accept the same language as D. However, observe that this is

somewhat redundant. That is, D does *not* need to diverge from M infinitely often. In fact, we only need D to behave differently at least once, and our proof would have still held. This is the essence of lazy diagonalization and represents our first key observation for our proof of the nondeterministic time hierarchy theorem.

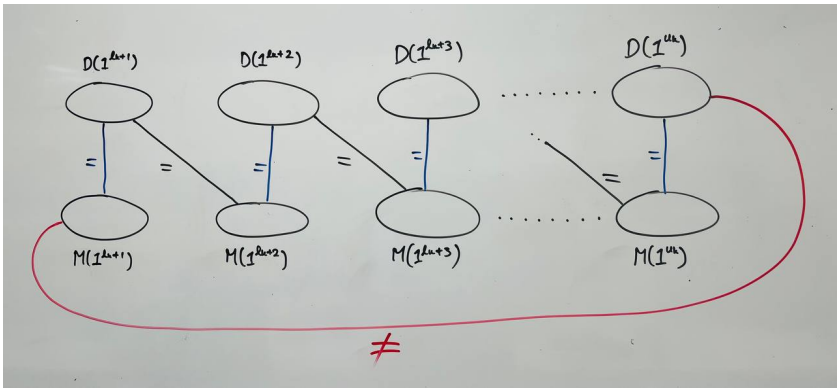
Next, our second key observation comes from the fact that we can "feed" M with an input that is much smaller than the original input. When D receives the input $\mathtt{1}^n$, rather than simulating M on $\mathtt{1}^n$, we can opt to simulate an exponentially smaller input, say $\mathtt{1}^{\log n}$, on M instead. Why would we want to do this? Currently, we have no effective means of deciding if M rejects some input w. Our only reliable strategy for now is to exhaustively check every computation path of M. But M may have exponentially many such paths. (Recall that this was the main roadblock that we faced earlier.) Consider what happens when we simulate $\mathtt{1}^{\log n}$ on M. If simulating one path of M takes $|\mathtt{1}^{\log n}| = O(\log n)$ time, then simulating all computation paths would require $2^{O(\log n)}$ time, which is $\text{poly}(n)$. Notice that $\text{poly}(n)$ is polynomial in the length of our original input string $\mathtt{1}^n$. This means that D won't run out of time while simulating M!

The set of all Turing machines is enumerable. Let M_k be the k-th Turing machine in the enumeration. We can associate each TM M_k with an interval $I_k = (l_k, u_k]$ (of natural numbers, i.e. $\mathbb{N}$). These intervals should be disjoint but contiguous. Each interval I_k is large, so we get to simulate M_k on a lot of input words. Crucially, the upper bound of the interval, u_k, should at the very least be exponentially larger than the lower bound, l_k.

Here's where the magic happens. Consider what happens when we run D on all the inputs that correspond to a single interval $I_k = (l_k, u_k]$ (thus, a single TM M_k). When D receives the input $\mathtt{1}^n$ where $n < u_k$, D accepts $\mathtt{1}^n$ iff M_k accepts $\mathtt{1}^{n+1}$. Here, we refrain ourselves from diagonalizing. However, when D receives the input $\mathtt{1}^n$ where $n = u_k$, we diagonalize but on a much smaller input, i.e. D accepts $\mathtt{1}^n$ iff M_k rejects $\mathtt{1}^{l_k+1}$. In this case, D has sufficient time to simulate M_k as we have established earlier. Furthermore, we have effectively demonstrated that no TM M_k can accept the same language as D, since D diverge from M_k on exactly one input, i.e. on the input $\mathtt{1}^n$ where $n = u_k$. To further illustrate this fact, consider the following graphical representation of an equivalence class and a non-equivalence relation that we obtain, by virtue of our construction.



If we assume the contrary that M_k accepts the same language as D, then we get the following new equivalence class.



This means that we have $M(1^{l_k+1}) = D(1^{u_k})$ by —— and ——, but $M(1^{l_k+1}) \neq D(1^{u_k})$ by ——. A contradiction.

One last technical point. Just as in the proof of the deterministic time hierarchy theorem, we may encounter a similar issue where the asymptotic behavior of f has not yet taken effect. To fix this problem, it suffices to simply restrict the simulation of D on sufficiently long inputs. We then associate the first(s) of these inputs to the first interval, I_1. □

Now, we describe the proof formally.

Proof. We know that the set of all Turing machines is enumerable. Let M_k be the k-th Turing machine in the enumeration. We associate M_k with an interval $I_k = (l_k, u_k]$, where $f(u_k) > 2^{f(l_k+1)^2}$. These intervals should be disjoint but contiguous. Because $f(n+1)$ is $o(g(n))$, some constant n_0 exists where $d \cdot f(n+1) < g(n)$ for all $n \geq n_0$. We set the lower bound of the first interval, l_1, to $f(n_0)$. Our NTM D will try to flip the answer of M_k only on one input in the interval I_k. We define D as follows:

D = On input w:

1. Let n be the length of w.
2. If w is not of the form 1^n where $n \geq n_0$, *reject*.
3. Compute k such that n lies in the interval $I_k = (l_k, u_k]$.
   - 3.1. If $n < u_k$, then nondeterministically simulate M_k on the input 1^{n+1} for up to $g(n)$ steps. If M accepts, then *accept*. If M rejects, then *reject*. (Observe that here we are *not* diagonalizing.)
   - 3.2. If $n = u_k$, then deterministically decide if M_k accepts the input 1^{l_k+1} by simulating all computation paths, for up to $g(n)$ steps in total. If M accepts, then *reject*. If M rejects, then *accept*. (Observe that here we are *indeed* diagonalizing. But we are doing it on a much shorter input than the input to D.)

Let's analyze the running time of each stage of D. Stage 1 and 2 clearly operate in $O(g(n))$ time. Stage 3 can also be performed in $O(g(n))$ time, since $k \ll n$. In stage 3.1, D nondeterministically simulates M_k on the input 1^{n+1}. If M_k really runs within $O(f(n+1))$ time, then D has sufficient time for the simulation, since $f(n+1) = o(g(n))$. Otherwise, if M_k doesn't halt, D will terminate the simulation after $g(n)$ steps. In both cases, D runs within $O(g(n))$ time. In stage 3.2, D deterministically simulates every

computation path of M_k on the input 1^{l_k+1}. If M_k really runs within $O(f(l_k+1))$, D again has sufficient time for the simulation. The deterministic simulation of each computation path requires $O(f(l_k+1)\log f(l_k+1))$, leading to a total time for all paths of $2^{O(f(l_k+1)\log f(l_k+1))}$. This is less than $2^{f(l_k+1)^2} < f(u_k) = f(n) = o(g(n))$. Otherwise, if M_k doesn't halt, D will stop the simulation after $g(n)$ steps. In both cases, D runs within $O(g(n))$ time.

Now, we show that A is not decidable in $O(f(n))$ time. Assume the contrary that some TM M_k decides A in time $f(n)$, where $f(n+1)$ is $o(g(n))$. Here, D can simulate M_k, using time $d \cdot g(n)$ for some constant d. Because $f(n+1)$ is $o(g(n))$, some constant n_0 exists where $d \cdot f(n+1) < g(n)$ for all $n \geq n_0$. Therefore, D's simulation of M_k will run to completion as long as the input has length n_0 or more. Consider what happens when we run D on all the inputs 1^n where $n \geq n_0$, such that all these inputs correspond to the interval $I_k = (l_k, u_k]$. Stages 3.1 and 3.2 of D ensure, respectively, that:

$$M_k(1^{n+1}) = D(1^n) \qquad \text{if } l_k < n < u_k \tag{1.1}$$

$$M_k(1^{l_k+1}) \neq D(1^{u_k}) = D(1^n) \qquad \text{if } n = u_k \tag{1.2}$$

By our assumption, D and M_k agree on all inputs 1^n in the interval $I_k = (l_k, u_k]$. Together with (2.1), this implies $M(1^{l_k+1}) = D(1^{l_k+1}) = M(1^{l_k+2}) = D(1^{l_k+2}) = \cdots = M(1^{u_k-1}) = D(1^{u_k-1}) = M(1^{u_k}) = D(1^{u_k})$. This contradicts $M(1^{l_k+1}) \neq D(1^{u_k})$ in (2.2). Hence, we conclude that A cannot be decided in $O(f(n))$ time. □

Remark 1.2.3 — Notice that the $1/\log f(n)$ factor does not appear in our hierarchy theorem, unlike in its deterministic counterpart. This absence is due to the efficient simulation of an NTM by another NTM, which only incurs a constant factor to the time complexity (see Exercise 2.6 of Arora and Barak [ArBa09]).

Remark 1.2.4 — What about the term "+1" in $f(n+1)$, where does it come from? This overhead arises from the fact that we have to simulate a longer input most of the time. That is, when we receive the input 1^n, we often simulate 1^{n+1} on M_k. We require that D has sufficient time to simulate M_k, so we need $f(n+1)$ to be $o(g(n))$.

Remark 1.2.5 — In other literature, one would often find that a rapidly growing function $h : \mathbb{N} \to \mathbb{N}$ is introduced in the proof. For instance, in Arora and Barak [ArBa09], the function h is defined as follows:

$$h(n) = \begin{cases} 2 & \text{if } n = 1 \\ 2^{h(n-1)^{1.2}} & \text{otherwise} \end{cases}$$

The interval I_k is then defined as $(l_k, u_k]$ where $l_k = h(k)$ and $u_k = h(k+1)$. It's important to note that such a proof only works for specific instances of the theorem, e.g. proving that $\mathbf{NTIME}(n) \subsetneq \mathbf{NTIME}(n^{1.5})$. For the proof to work in the general case, we must define h in relation to f, as demonstrated in our proof.

Next, let us present a different proof due to Fortnow and Santhanam [FoSa11].

Proof Idea. Once again, we define a language A by describing a machine D that decides it. We begin with an input w of the form $\langle M \rangle 01^k0$, where k is sufficiently large. How

large? Well, large enough such that the sequence of 1s acts as a padding, allowing the asymptotic behavior to kick in. Just like in our previous proof, we use D to simulate some TM M. Most of the time, we want to feed M with a slightly longer input. If we receive $\langle M \rangle \, 01^k0$ for instance, we simulate M on the inputs $\langle M \rangle \, 01^k00$ and $\langle M \rangle \, 01^k01$. We're not in a rush to diagonalize here, so D will only accept if M accepts on both of those inputs. We can generalize this idea: for an input of the form $\langle M \rangle \, 01^k0y$ where $y = (0+1)^*$, we simulate M on $\langle M \rangle \, 01^k0y0$ and $\langle M \rangle \, 01^k0y1$. We continue this process, by simulating M on longer and longer inputs, until we reach an input $\langle M \rangle \, 01^k0y$ such that y is "long enough". Once y is long enough, instead of continuing to simulate longer inputs, we switch to simulating our initial, shorter input $\langle M \rangle \, 01^k0$, using y as advice—basically, treating it as the nondeterministic choices. Then, we diagonalize, so D accepts $\langle M \rangle \, 01^k0y$ iff M rejects $\langle M \rangle \, 01^k0$ with y as advice.

Finally, we finish with the standard *reductio ad absurdum* (proof by contradiction) arguments. To make this work, the advice y must be long enough to simulate the longest computation path of M. Thus, it suffices to set the constraint $y \geq g(|\langle M \rangle| + k + 2)$ as a prerequisite for the diagonalization. At the end, we will discover that $\langle M \rangle \, 01^k0 \in A$ iff M rejects $\langle M \rangle \, 01^k0$ on all computation paths. But M is supposed to decide A. This is a contradiction. □

Now, we lay out the proof more formally. To make things clearer, instead of using our original encoding of the input, i.e. $\langle M \rangle \, 01^k0y$, we will rewrite it in a more mathematical notation such as $(M, 1^k, y)$. This should help make the proof easier to follow.

Proof. The following $O(g(n))$ time NTM D decides a language A that is not decidable in $O(f(n))$ time.

D = On input $w = (M, 1^k, y)$

1. Let n be the length of w.
2. If $|y| < g(|M| + k)$, then simulate $M(M, 1^k, y0)$ and $M(M, 1^k, y1)$ for at most $g(n)$ steps each, and *accept* iff they both accept.
3. If $|y| \geq g(|M| + k)$ then *accept* iff $M(M, 1^k, \epsilon)$ rejects using y as advice (i.e. nondeterministic choices). Note that if M does not halt on this computation path (e.g. because y is not long enough), then *reject*.

It is easy to see that D runs within $O(g(n))$ time. Now, we show that A is not decidable in $O(f(n))$ time. Assume the contrary that some NTM M_A decides A in $O(f(n))$ time. Consider an input of the form $w = (M_A, 1^k, \epsilon)$ for sufficiently large k. We have:

$$
\begin{aligned}
w \in A &\iff (M_A, 1^k, \epsilon) \in A && \text{(because } w = (M_A, 1^k, \epsilon)\text{)} \\
&\iff M_A(M_A, 1^k, 0) = M_A(M_A, 1^k, 1) = 1 && \text{(by definition of } D\text{, stage 2)} \\
&\iff (M_A, 1^k, 0), (M_A, 1^k, 1) \in A && \text{(because } M_A \text{ decides } A\text{)} \\
&\iff M_A(M_A, 1^k, 00) = M_A(M_A, 1^k, 01) = 1 \\
&\qquad M_A(M_A, 1^k, 10) = M_A(M_A, 1^k, 11) = 1 && \text{(by definition of } D\text{, stage 2)}
\end{aligned}
$$

Let t be the smallest integer such that $t \geq g(|M_A| + k)$. Continuing the above line of

reasoning we get:

$$
\begin{aligned}
w \in A &\iff \forall y \in \{0,1\}^t : M_A(M_A, 1^k, y) = 1 && \text{(by induction)} \\
&\iff \forall y \in \{0,1\}^t : (M_A, 1^k, y) \in A && \text{(because } M_A \text{ decides } A\text{)} \\
&\iff \forall y \in \{0,1\}^t : D(M_A, 1^k, y) = 1 && \text{(because } D \text{ decides } A\text{)} \\
&\iff \forall y \in \{0,1\}^t : M_A(M_A, 1^k, \epsilon) \text{ rejects} \\
&\qquad\qquad \text{on computation path } y && \text{(by definition of } D\text{, stage 3)} \\
&\iff M_A(M_A, 1^k, \epsilon) = 0 && \text{(by the reject condition of NTMs)} \\
&\iff (M_A, 1^k, \epsilon) \notin A && \text{(because } M_A \text{ decides } A\text{)} \\
&\iff w \notin A && \text{(because } w = (M_A, 1^k, \epsilon)\text{)}
\end{aligned}
$$

A contradiction. We conclude that no such machine M_A can exist. □

Remark 1.2.6 — One advantage of this proof over Žák's is that we only need $f(n)$ steps for the diagonalization instead of exponential in $f(n)$. Also, we don't need to appeal to some definition of a rapidly growing function. However, one drawback is that it's more difficult to generalize this proof to a broader set of complexity classes.

§1.3 Ladner's Theorem: Existence of NP-Intermediate Problems

One striking aspects of **NP**-completeness is that a surprisingly large number of **NP** problems—including some that were studied for many decades—turned out to be **NP**-complete. (See the appendix of Garey and Johnson [GaJo79] for a well-curated list of 320 **NP**-complete problems.) This phenomenon suggests a bold conjecture: Every problem in **NP** is either in **P** or **NP**-complete. If $\mathbf{P} = \mathbf{NP}$, then the conjecture is trivially true but not interesting. In this section, we show that if $\mathbf{P} \neq \mathbf{NP}$, then this conjecture is false, i.e. there is a language $L \in \mathbf{NP} \setminus \mathbf{P}$ that is not **NP**-complete.

Definition 1.3.1 (**NP**-intermediate languages)

An **NP-intermediate** language is a language that is in **NP** but is neither in **P** nor is it **NP**-complete.

Theorem 1.3.1 (Ladner's theorem)

If $\mathbf{P} \neq \mathbf{NP}$, then there exists an **NP**-intermediate language.

Remark 1.3.2 — In fact, Ladner [La75] showed a stronger result, that there is an infinite hierarchy between **P** and **NP**-complete, assuming $\mathbf{P} \neq \mathbf{NP}$.

Let us explore two different proofs of Ladner's Theorem. The first proof works by padding an **NP**-complete language, while the second involves "blowing holes" in an **NP**-complete language. We will start with the first one, because I think it's a bit more intuitive.

§1.3.1 Proof by padding an NP-complete language

This proof is based on an unpublished manuscript by Russell Impagliazzo. The following presentation draws from the excellent lecture notes by Ben-David [Be21]. Just a heads up, we will be diving deep into the intuition behind the proof, so the following exposition is going to be lengthy. We will also take some time to formalize the proof and work through the technical details. To round out our discussion, I will offer a summarized version of the proof at the end, highlighting only the key points.

Padding

To begin the proof, we will start with a language C which is **NP**-complete (e.g. SAT), and then pad it to create an easier language A. We define A to be

$$A = \{x\texttt{01}^{f(|x|)} \mid x \in C\}$$

for some function $f : \mathbb{N} \to \mathbb{N}$. That is, A contains all the strings in C, but with $f(n)$ additional `1`s appended, where n is the length of the string that was in C.

We've seen padding before. The idea of it is that the padded language A is essentially equivalent to the original language C, but its inputs are longer. Since we measure the amount of resources we use in terms of its input size, it will be "easier" to compute A than to compute C, because our time budget will be larger. For example, to show that $A \in \mathbf{P}$, we would merely need an algorithm that decides whether $x \in C$ in time that is polynomial in $|x\texttt{01}^{f(|x|)}| = |x| + 1 + f(|x|)$, rather than polynomial in $|x|$. If $f(|x|)$ is large, solving A becomes much easier than finding a polynomial-time algorithm for C.

Our main trick for the proof is to choose f in such a way that A has the desired property (of being an **NP**-intermediate language). We will ensure that f is non-decreasing and time-constructible, though the specific choice of f will have to wait a bit.

The padded language is in NP

We argue that A is in **NP**.

Proof. Recall that we chose C to be in **NP**, so C has a polynomial-time verifier V_C. We now construct a polynomial-time verifier V_A for A. On input $\langle y, w\rangle$, V_A will check if y is of the form $y = x\texttt{01}^{f(|x|)}$ for some string x. Checking this requires computing $f(|x|)$, but that takes at most $O(f(|x|))$ time if f is time-constructible, so this is linear time in the input size. If the input is badly formatted, V_A will reject.

If the input is in the right format, V_A will then simulate $V_C(\langle x, w\rangle)$, and output what V_C outputs. Note that the string $\langle x, w\rangle$ is shorter than the input string $\langle y, w\rangle$ to V_A. Since V_C runs in polynomial time, it follows that V_A runs in polynomial time. Now, if $y \in A$, then $y = x\texttt{01}^{f(|x|)}$ for some string $x \in C$, and so some witness w of size polynomial in $|x|$ exists such that $V_C(\langle x, w\rangle)$ accepts; in this case, $V_A(\langle y, w\rangle)$ will accept. Conversely, if $y \notin A$, then either y is not formatted like $x\texttt{01}^{f(|x|)}$ for any string x, or else it is formatted this way but x is not in C. Either way, there is no witness w that causes $V_A(\langle y, w\rangle)$ to accept. We conclude that V_A is a polynomial-time verifier for A, so A is in **NP**. □

This proof only required that $C \in \mathbf{NP}$ and that the function f was time-constructible.

If f is polynomial

What happens if we choose f to be a polynomial function? In this case, we claim that $C \leq_m^P A$, that is, C Karp-reduces to A.

Proof. To show this, we need a polynomial-time mapping reduction from C to A. We describe a machine M to implement this reduction. Given an input x, the machine M will compute $f(|x|)$ and then output $x\mathtt{01}^{f(|x|)}$. Since f is time-constructible, this takes $O(f(|x|))$ time. Since we've chosen f to have polynomial growth rate, the running time of M is also polynomial. Also, if $x \in C$ then $x\mathtt{01}^{f(|x|)} \in A$ and if $x \notin C$ then $x\mathtt{01}^{f(|x|)} \notin A$ by the definition of A. Hence M is a polynomial-time mapping reduction, so $C \leq_m^P A$. □

Note that we won't necessarily choose f to grow polynomially, but if we do (and if it's time-constructible), it will imply that $C \leq_m^P A$.

If f is super-polynomial

What happens if we choose f to grow super-polynomially? That is, $f(n) = \Omega(n^k)$ for all $k \in \mathbb{N}$, so f grows faster than any polynomial. In this case, we claim that there is no polynomial-time mapping reduction from C to A.

Proof Idea. To see this, we assume the contrary that we had a polynomial-time mapping reduction g from C to A. From this, we will show that we can construct another function h' that serves as a polynomial-time mapping reduction from C to some finite set F. Since every finite set is in $\mathbf{P}$ (as a Turing machine can simply hard-code all the answers using its internal states), this implies that C can be reduced in polynomial time to a language in $\mathbf{P}$, which means $C \in \mathbf{P}$. This contradicts our assumption that $\mathbf{P} \neq \mathbf{NP}$. (Because then all $\mathbf{NP}$-complete languages such as C cannot be in $\mathbf{P}$.) □

Proof. Suppose we had a polynomial-time mapping reduction g from C to A. We first claim that we would also have a polynomial-time mapping reduction g' from C to A such that for each string y, $g'(y)$ is of the form $x\mathtt{01}^{f(|x|)}$ for some string x. To see this, we define a Turing machine M computing g' as follows:

$$M(y) = \begin{cases} g(y) & \text{if } y \in C \\ g(y) & \text{if } y \notin C \text{ and } g(y) \text{ is correctly formatted} \\ z\mathtt{01}^{f(|z|)} & \text{if } y \notin C \text{ and } g(y) \text{ is badly formatted} \end{cases}$$

Let us describe M briefly. The first step of M on input y will be to compute $g(y)$ (which takes polynomial time in $|y|$). Next, M will check $g(y)$ is formatted like $x\mathtt{01}^{f(|x|)}$ for some string x. This requires computing $f(|x|)$ but f is time-constructible, so that takes $O(f(|x|))$ and the whole check can be done in linear time. If the formatting check passes, M will output $g(y)$. Otherwise, M will output $z\mathtt{01}^f(|x|)$, where z is a fixed string not in C. The string z will be hard-coded into M, and printing out $z\mathtt{01}^{f(|z|)}$ will take constant time as z is a fixed, constant string.

It is clear that M runs in polynomial time. Moreover, if $y \in C$, then $g(y) \in A$, as g is a mapping reduction from C to A; but this means that $g(y)$ is formatted like $x\mathtt{01}^{f(|x|)}$, so $M(y)$ outputs $g(y)$, which is in A. On the other hand, if $y \notin C$, then $g(y) \notin A$, and there are two options, either $g(y)$ is badly formatted, in which case $M(y)$ outputs $z\mathtt{01}^{f(|z|)}$,

or else it is correctly formatted, in which case $M(y)$ outputs $g(y) = x\texttt{01}^{f(|x|)}$ for some string x not in C. This confirms that M computes a polynomial-time mapping reduction g' that gives strings of the desired format.

Let h denote the function that maps y to the string x for which $g'(y) = x\texttt{01}^{f(|x|)}$. Then h is computable in polynomial-time, and for all strings y, we have $y \in C$ iff $h(y) \in C$. Next, note that since g' is computable in polynomial time in $|y|$, the size $|g'(y)|$ must grow polynomially with $|y|$. Hence, for sufficiently large y, we have $f(|y|) > |g'(y)|$ (since f grows faster than any polynomial). This means that there is some number $n_0 \in \mathbb{N}$ such that for all strings y with $|y| \geq n_0$, we have $|g'(y)| < f(|y|)$. Since $g'(y) = x\texttt{01}^{f}(|x|)$ for some x, we must have $|x| + 1 + f(|x|) < f(|y|)$ whenever $|y| \geq n_0$. This implies that when $|y| \geq n_0$, we have $f(|x|) < f(|y|)$, and since f is non-decreasing, $|x| < |y|$. In other words, for all strings y with $|y| \geq n_0$, we have $|h(y)| < |y|$ (because $x = h(y)$).

Now consider the Turing machine which takes input y and applies h to y repeatedly, for a total of $|y|$ times. Since each application of h results in a shorter string if $|y| \geq n_0$, or else results in a string of length at most n_0 if $|y| < n_0$, we conclude that each application of h can be computed in polynomial time in $|y|$ (since n_0 is a constant). Moreover, we are doing only $|y|$ such computations. Hence this Turing machine runs in polynomial time in the input size $|y|$. Let h' be the function computed by this Turing machine.

Then for every string y, we have $h'(y) \in C$ iff $y \in C$. Moreover, since each application of h decreases the size of y by at least one unless $|y| < n_0$, it must be the case that $|h'(y)| \leq n_0$ for all y. Therefore, in polynomial time, we can compute a function $h'(y)$ which maps y to a string of length of at most n_0 such that $h'(y)$ is in C if and only if $y \in C$. In other words, h' is a polynomial-time mapping reduction from C to $C \cap \{y \mid |y| \leq n_0\}$. But the latter is a finite set! Every finite set is in $\mathbf{P}$ because we can create a Turing machine that has all the answers hard-coded using the internal states. So we have a polynomial-time mapping reduction from C to a language in $\mathbf{P}$, and hence $C \in P$. This contradicts our assumption that $\mathbf{P} \neq \mathbf{NP}$. Hence, there is no polynomial-time mapping reduction from C to A. □

This proof only required that f is non-decreasing, time-constructible, and grows faster than any polynomial.

The desired choice of f

Here's what we want to do. We want to choose f so that it grows polynomially if $A \in \mathbf{P}$ and super-polynomially if $A \notin \mathbf{P}$. Why? Consider the following arguments. There are only two possibilities, either $A \in \mathbf{P}$ or $A \notin \mathbf{P}$.

1. If $A \in \mathbf{P}$, then f grows polynomially, and so $C \leq_m^P A$. This implies that $C \in \mathbf{P}$, giving a contradiction under our assumption that $\mathbf{P} \neq \mathbf{NP}$. This will therefore mean that $A \notin \mathbf{P}$.
2. If $A \notin \mathbf{P}$, then f grows super-polynomially, and so there is no polynomial-time mapping reduction from C to A. Since C is $\mathbf{NP}$-complete, this implies that A is not $\mathbf{NP}$-complete. This will force A to be $\mathbf{NP}$-intermediate.

Essentially, we've designed f in a way that forces A to be $\mathbf{NP}$-intermediate. If we could construct f to behave like this (and ensure that f is non-decreasing and time-constructible), we would be done. The problem is that the definition of A depends on f, so this seems circular: the choice of f depends on A, and the definition of A depends on

f. To resolve this, we will do something that essentially amounts to a diagonalization argument.

The diagonalization argument

We now present and formalize the diagonalization argument. First, let $M_1, M_2, \ldots$ be an enumeration of Turing machines. We assign a restriction (or intuitively, an "alarm") to each M_i such that the running time of $M_i(x)$ is bounded by $|x|^i + i$ steps. (The additional term "$+i$" accounts for the length of the encoding for M_i. Recall that every natural number (in unary) $\mathtt{1}^i$ can be interpreted as some TM M_i.) We claim that $M_1, M_2, \ldots$ is an enumeration of all polynomial-time algorithms.

Proof. Let $p : \mathbb{N} \to \mathbb{N}$ be an arbitrary polynomial function, i.e. $p(n) = O(n^k)$ for some $k \in \mathbb{N}$. It is easy to see that there exists a smallest $i \in \mathbb{N}$ such that $n^i + i \geq p(n)$. This implies that for any $j \geq i$, the machine M_j in the enumeration will have a running time that is at least as large as $p(n)$, and thus can represent the polynomial function p. □

We define f by describing a Turing machine M_f that computes it. To be time-constructible, we want $M_f(\mathtt{1}^n)$ to output $f(n)$ in $O(f(n))$ time. M_f is a recursive algorithm[1], meaning $f(n)$ is defined inductively, so that $f(n)$ depends on the values of $f(k)$ for $k < n$. We start by setting some base cases such as $f(0) = 0$ and $f(1) = 1$. Next, on input $\mathtt{1}^n$, the machine will go through the first n Turing machines $M_1, M_2, \ldots, M_n$ (from our enumeration) in order. For each machine M_i, M_f will attempt to disprove the claim "M_i decides A in polynomial time". (We will soon discuss how M_f will do this.) Then M_f keep track of the smallest i for which this claim was not disproven, and it will output n^{i+3}. The idea is that if some Turing machine M decides A in polynomial time, then for some integer i we will have $M_i = M$, and hence for all $n \geq i$, the machine M_f will fail to disprove that M_i decides A, meaning that $M_f(\mathtt{1}^n)$ will return n^{i+3} for all sufficiently large n; this will cause f to grow polynomially if A is in $\mathbf{P}$. On the other hand, if no Turing machine decides A in polynomial time, then eventually, M_f should disprove the claim "M_i decides A in polynomial time" for every given i, and hence f will grow faster than n^{i+3} for every $i \in \mathbb{N}$; this will cause f to grow super-polynomially if A is not in $\mathbf{P}$.

Before we describe M_f in more detail, we will introduce the machine M_C which decides C in exponential time. This machine exists since $\mathbf{NP} \subseteq \mathbf{EXPTIME}$, which means that since $C \in \mathbf{NP}$ we have $C \in \mathbf{EXPTIME}$. In particular, we have $C \in \mathbf{DTIME}(2^{n^k})$ for some $k \in \mathbb{N}$. We let M_C be a Turing machine which decides C and has running time $O(2^{n^k})$. Back to M_f. On input $\mathtt{1}^n$, its first step is to calculate $l = (\log n)^{1/(k+1)}$. This number l is sufficiently small that simulating M_C on an input string of size at most l can be done in $O(n)$ time. This calculation of l can actually be done in $O(n)$ time (first M_f calculates n in binary from its input $\mathtt{1}^n$, and then it can perform computations on n that are even allowed to take exponential time, and this will still take only $O(n)$ time because the binary representation of n takes $O(\log n)$ space). Note that there are also only at most $O(n)$ strings of length at most l.

Next, M_f will write down the values of $f(k)$ for all $k \leq l$ by recursively calling itself on smaller inputs. We will soon show that M_f will only take $O(n^3)$ time for all its calculations other than this recursive step; by using dynamic programming (e.g. computing $f(0)$, then $f(1)$, then $f(2)$, etc., and looking these up in memory instead of making recursive calls), we only make l total calls to M_f on such inputs $k \leq l$, each with input $\mathtt{1}^l$ or less.

[1] From the familiar notion of recursion in programming that we all adore.

This means the total time to compute all the values of $f(k)$ for $k \leq l$ will be $O(l^4)$, which is smaller than $O(n)$.

The next step of M_f will be to iterate over all strings s of length at most l; recall that there are at most $O(n)$ such strings. For each such string s, M_f will decide whether s is in A. To do so, M_f will need to check if s looks like $x\mathtt{01}^{f(|x|)}$ for some string x; luckily, M_f already wrote down the values of $f(k)$ for all $k \leq l$, so this check is easy and can be done in linear time in l. If the formatting check passes, M_f will also need to check whether x is in C, which it can do by simulating $M_C(x)$; recall this can be done in $O(n)$ steps. Therefore, after $O(n^2)$ total steps, M_f will produce a table that lists, for each string s of length at most l, whether s is in A.

The next step of M_f will be to iterate over the first n Turing machines $M_1, M_2, \ldots, M_n$ in order. It takes only $O(n)$ time to list them. For each machine, M will again iterate over all strings of length at most l. Then for each pair of machine M_i and string s with $|s| \leq l$, the machine M_f will simulate $M_i(s)$ for n steps. If $M_i(s)$ halts in that time, M_f will check whether it correctly decides if $s \in A$ (by consulting the table). M_f will then find the smallest number j such that some machine M_i for $i \leq j$ correctly decided if $s \in A$ for all strings s of length at most l. The running time of M_i on each string s was at most $|s|^i + i \leq |s|^j + j$. If there is no such j (either because none of the M_i gave the correct answers, or this property wasn't properly checked due to the time constraint of n steps), M_f will set j to n. Then M_f will output n^{j+3}.

Since there are n machines and at most $O(n)$ strings, all simulations together will take $O(n^3)$ steps, and hence M_f runs in $O(n^3)$ time. Additionally, it should be clear that f is increasing, because the value of j that M_f finds can only increase. It is also clear that $f(n) = \Omega(n^3)$, and hence f is time-constructible, since M_f outputs $f(n)$ in at most $O(f(n))$ time.

Now, we show that our construction of M_f forces A to be **NP**-intermediate.

Proof. If $A \in \mathbf{P}$, it means that there are some Turing machine M_i that decides A in polynomial time. The running time of M_i is polynomial, so for sufficiently large j, it takes M_i fewer than $|s|^j + j$ steps to halt on each string s. Since M_i decides A, it therefore follows that for all sufficiently large $n \in \mathbb{N}$, we will have $f(n) = n^{\max(i,j)+3}$. Hence, f grows polynomially if $A \in \mathbf{P}$, and as we've seen, that causes $C \leq_m^P A$ to hold, so $C \in \mathbf{P}$, which gives a contradiction. From this we conclude that $A \notin \mathbf{P}$.

Next, if $A \notin \mathbf{P}$, then for every machine M_i and every $j \in \mathbb{N}$, there will be some string s such that either $M_i(s)$ gives the wrong answer as to whether $s \in A$, or else $M_i(s)$ takes more than $|s|^j + j$ steps. This means that there will not be any $j \in \mathbb{N}$ such that $f(n) \leq n^j + j$ for all $n \in \mathbb{N}$. From this it follows that $f(n)$ cannot be upper bounded by any polynomial function, since every polynomial function can be upper bounded by a function of the form $n^j + j$. In other words, $f(n)$ grows super-polynomially, and as we've seen, this implies that there is no polynomial-time mapping reduction from C to A. Since C is **NP**-complete, it follows that A is not **NP**-complete, so it is **NP**-intermediate, as desired. □

A summary of the proof

Let us summarize the key points of our proof here. We start with a language C which is **NP**-complete. Let M_C be a Turing machine which decides M_C and has running time

$O(2^{n^k})$ for some $k \in \mathbb{N}$. Now, we pad C to create an easier language A. We define A to be

$$A = \{x\mathtt{01}^{f(|x|)} \mid x \in C\}$$

for some function $f : \mathbb{N} \to \mathbb{N}$. The choice of f is crucial. Here, we want f to grow polynomially if $A \in \mathbf{P}$ and super-polynomially if $A \notin \mathbf{P}$. This behavior of f forces A to be **NP**-intermediate. Why? There are only two possibilities, either $A \in \mathbf{P}$ or $A \notin \mathbf{P}$.

1. If $A \in \mathbf{P}$, then f grows polynomially, and so $C \leq_m^P A$. This implies that $C \in \mathbf{P}$, giving a contradiction under our assumption that $\mathbf{P} \neq \mathbf{NP}$. This will therefore mean that $A \notin \mathbf{P}$.
2. If $A \notin \mathbf{P}$, then f grows super-polynomially, and so there is no polynomial-time mapping reduction from C to A. Since C is **NP**-complete, this implies that A is not **NP**-complete. This will force A to be **NP**-intermediate.

Let $M_1, M_2, \ldots$ be an enumeration of polynomial-time Turing machines, where each machine M_i runs for at most $|x|^i + i$ steps on input x. Such an enumeration can be constructed quite easily.

We define f by giving a Turing machine M_f that computes it.

M_f = On input $\mathtt{1}^n$:

1. Compute $l = (\log n)^{1/(k+1)}$, where k comes from the definition of M_C.
2. Compute the values $f(0), f(1), \ldots, f(i)$ where $i \leq l$. This can be done efficiently by calling $M_f(\mathtt{1}^i)$ in order from $i = 0$ to l, and storing all the values in memory.
3. Iterate over all strings s of length at most l. For each string s, decide whether s is in A. To do so, check if s looks like $x\mathtt{01}^{f(|x|)}$ for some string x. We already have the values of $f(k)$ for all $k \leq l$, so this formatting check is easy. Then, to check if x is in C, simulate $M_C(x)$. Produce a table that lists, for each string s of length at most l, whether s is in A.
4. Iterate over the first n Turing machines $M_1, \ldots, M_n$ in order. For each machine M_i, iterate over all strings of length at most l. For each pair of machine M_i and string s with $|s| \leq l$, simulate $M_i(s)$ for n steps. If $M_i(s)$ halts in that time, check whether M_i correctly decides if $s \in A$ by consulting the table. Find the smallest j such that some M_i for $i \leq j$ gives the correct answers for all strings s. If there is no such j, set j to n.
5. Output n^{j+3}.

It is easy to see that $f(n) = \Omega(n^3)$, and hence f is time-constructible, since M_f outputs $f(n)$ in at most $O(f(n))$ time. We now show that M_f computes a function f that exhibit our desired property, thus showing that $A \in \mathbf{NP}$-intermediate.

Proof. If $A \in \mathbf{P}$, there must exist some Turing machine M_j (where $j \in \mathbb{N}$) that decides A in polynomial time. In this case, M_f will output a function f that is "fixed" with respect to j. Specifically, for all sufficiently large $n \in \mathbb{N}$, we get $f(n) = n^{j+3}$ where j is some fixed constant. This shows that f grows polynomially. However, this leads to a contradiction, as we've previously shown. Therefore, we conclude that $A \notin \mathbf{P}$. As a

result, there cannot be any M_j that decides A in polynomial time. This forces M_f to compute a function f such that $f(n)$ cannot be bounded by any $n^j + j$ for some fixed constant j. This is because, as n increases, j must also increase—in fact, j increases linearly with n by definition of M_f. Therefore, f grows faster than any polynomial, which as we've already established, places A in the **NP**-intermediate class. □

§1.3.2 Proof by blowing holes in an NP-complete language

This proof was originally due to Ladner [La75]. The presentation below follows a later paper by Downey and Fortnow [DoFo03].

Write the rest of this.

§1.3.3 Natural candidates for NP-intermediate

If **P** ≠ **NP**, Ladner's theorem demonstrates the existence of **NP**-intermediate sets, one that lie strictly between **P** and the **NP**-complete sets. Indeed, by applying the construction of the proof iteratively, it is possible to show that there is an entire infinite chain of **NP**-intermediate sets, each irreducible to the next, yet all in **NP** \ **P**. However, these sets are somewhat contrived, and it is reasonable to ask if we can point to any practically-motivated sets that might be **NP**-intermediate.

Ladner [La75] mentions three sets that were candidates for being **NP**-intermediate at the time he proved his theorem: PRIME (is x prime?), LINEAR INEQUALITIES (is a given linear program feasible?), and GRAPH ISOMORPHISM (are the two graphs G and H isomorphic?). All of these were known to be in **NP** as of 1975, and none of them were known to be in **P** or **NP**-complete, but there was no argument that any of them in particular were **NP**-intermediate if **P** ≠ **NP**. Since then, two of them (PRIME [AgKaSa04] and LINEAR INEQUALITIES [Kh80]) have been shown to be in **P**. The status of GRAPH ISOMORPHISM is still open, with the best known algorithm running in quasi-polynomial time [Ba16].

§1.4 Oracle Machines and the Limits of Diagonalization

Complexity theory is unusual as a field of mathematics in having a rich collection of barrier results that show that certain proof techniques cannot be used to solve core problems like **P** vs **NP**. One such technique is diagonalization. The magic of diagonalization is how abstract and general it is: it never requires us to "get our hands dirty" by understanding the inner workings of algorithms. But as was recognized early in the history of complexity theory, the price of generality is that the logical techniques are extremely limited in scope.

Often the best way to understand the limits of a proposed approach for proving a statement S, is to examine what else besides S the approach would prove if it worked, i.e. which stronger statements S' the approach "fails to differentiate" from S. If any of the stronger statements are false, then the approach can't prove S either.

That is exactly what Baker, Gill, and Solovay [BaGiSo75] did for diagonalization in 1975, when they articulated the **relativization barrier**. The central insight was that almost all the techniques we have for proving statements in complexity theory—such as $C \subseteq D$ or $C \not\subseteq D$, where C and D are two complexity classes—are so general that, if they work at all, then they can actually prove $C^A \subseteq D^A$ or $C^A \not\subseteq D^A$ *for all possible oracles* A. In

other words, if all the machines that appear in the proof are enhanced in the same way, by being given access to the same oracle, the proof is completely oblivious to that change, and goes through just as before. A proof with this property is said to "relativize", or to hold "in all possible relativized worlds" or "relative to any oracle."

Why do so many proofs relativize? Intuitively, because the proofs, as with the case of diagonalization, only do things like using one Turing machine M_1 to simulate a second Turing machine M_2 step-by-step, without examining either machine's internal structure. In that case, if M_2 is given access to an oracle A, then M_1 can still simulate M_2 just fine, provided that M_1 is also given access to A, in order to simulate M_2's oracle calls. To illustrate, you might want to check that the alternative proof of Corollary 1.1.8 can be easily modified to show that $\mathbf{P}^A \neq \mathbf{EXPTIME}^A$ for all oracles A.

Alas, Baker, Gill, and Solovay then observed that no relativizing technique can possibly resolve the **P** vs **NP** question. For, unlike **P** vs **EXPTIME**, or the unsolvability of the halting problem, **P** vs **NP** admits "contradictory relativizations." That is, there are some oracle worlds where $\mathbf{P} = \mathbf{NP}$, and others where $\mathbf{P} \neq \mathbf{NP}$. For this reason, any proof of $\mathbf{P} \neq \mathbf{NP}$ will need to "notice", at some point, that there are no oracles in "our" world: it will have to use techniques that *fail* relative to certain oracles.

§1.4.1 Oracles

We will soon discuss the main result of Baker, Gill, and Solovay. But before we do that, let us first formalize the notion of oracle machines in the following.

Definition 1.4.1 (Oracles)

An **oracle** for a language A is a device that is capable of reporting whether any string w is a member of A. Thus, formally an oracle is just a function $\mathcal{O} : \{0,1\}^* \to \{0,1\}$ (or equivalently, a set $\mathcal{O} \subseteq \{0,1\}^*$) that corresponds to some language A.

Definition 1.4.2 (Oracle Turing machines)

An **oracle Turing machine** M^A is a modified Turing machine that has the additional capability of querying an oracle. Whenever M^A writes a string on a special **oracle tape**, it is informed whether that string is a member of A, in a single computation step.

Formally, this can be done with the addition of three special states: q_{query}, q_{yes}, and q_{no}. Whenever during the execution M^A enters the state q_{query}, the machine moves into the state q_{yes} if $q \in A$ and q_{no} if $q \notin A$, where q denotes the contents of the oracle tape.

Nondeterministic oracle Turing machines are defined similarly.

Definition 1.4.3 (Oracle classes)

We define C^A to be the complexity class of decision problems solvable by an algorithm in class C with an oracle for a language A.

Example 1.4.1
$\mathbf{P}^{\mathrm{SAT}}$ is the class of problems solvable in polynomial time by a deterministic Turing machine with an oracle for SAT.

The notation can be extended to the set of languages. For instance, when X and Y are complexity classes, we write X^Y for the class of languages computable by a machine in class X using an oracle from class Y. Formally,

$$X^Y = \bigcup_{L \in Y} X^L.$$

When the language L is complete for some class Y, then $X^L = X^Y$ provided that machines in X can execute reductions used in the completeness definition of class Y. In particular, since SAT is **NP**-complete with respect to polynomial-time reductions, we have $\mathbf{P}^{\mathrm{SAT}} = \mathbf{P}^{\mathbf{NP}}$. However, $\mathbf{DLOGTIME}^{\mathrm{SAT}}$ may not equal $\mathbf{DLOGTIME}^{\mathbf{NP}}$.

Also, notice that the definition makes X^Y at least as powerful as the strongest of X and Y, and may give it even more power. For example, if $\mathbf{NP} \neq \mathbf{coNP}$, then $\mathbf{NP} \subsetneq \mathbf{P}^{\mathbf{NP}}$, since in $\mathbf{P}^{\mathbf{NP}}$ we can solve any problem in **NP** just by asking the oracle for its opinion, but we can also solve any problem in **coNP** by asking the oracle for its opinion and giving the opposite answer. And this is all without even taking advantage of the ability to ask multiple questions and have the later questions depend on the outcome of earlier ones.

§1.4.2 The Baker-Gill-Solovay theorem

Now, we state the result formally.

Theorem 1.4.2 (The Baker-Gill-Solovay theorem)
There exists an oracle A such that $\mathbf{P}^A = \mathbf{NP}^A$, and another oracle B such that $\mathbf{P}^B \neq \mathbf{NP}^B$.

Proof Idea. Exhibiting an oracle A is easy. Just let A be any **PSPACE**-complete problem such as TQBF. We exhibit oracle B by construction. We design B so that a certain language L_B in $\mathbf{NP}^B$ provably requires brute-force search, and so L_B cannot be in $\mathbf{P}^B$. Hence, we can conclude that $\mathbf{P}^B \neq \mathbf{NP}^B$. The construction considers every polynomial-time oracle machine in turn and ensures that each fails to decide the language L_B. (This is diagonalization!) □

Proof. Let A be TQBF. We have a series of containtments

$$\mathbf{NP}^{\mathrm{TQBF}} \overset{1}{\subseteq} \mathbf{NPSPACE} \overset{2}{\subseteq} \mathbf{PSPACE} \overset{3}{\subseteq} \mathbf{P}^{\mathrm{TQBF}}.$$

Containtment 1 holds because we can convert the nondeterministic polynomial-time oracle TM to a nondeterministic polynomial-space machine that computes the answers to queries regarding TQBF instead of using the oracle. Containtment 2 follows from Savitch's theorem. Containtment 3 holds because TQBF is **PSPACE**-complete. Hence, we conclude that $\mathbf{P}^{\mathrm{TQBF}} = \mathbf{NP}^{\mathrm{TQBF}}$.

Next, we show how to construct oracle B. For any oracle B, we define

$$L_B = \{1^n \mid B \text{ contains a string of length } n\}.$$

Obviously, for any B, L_B is in $\mathbf{NP}^B$. Here's why. A nondeterministic linear-time oracle Turing machine M that uses the oracle B can recognize L_B as follows. On the input 1^n of length n, M nondeterministically guesses a string x of length n, queries it on the oracle set B, and accepts if and only if $x \in B$. In particular, if $1^n \in L_B$, then one of the computation paths of M will guess a correct string x of length n in B so that computational path will accept 1^n. On the other hand, if $1^n \notin B$, then all strings x of length n guessed by M are not in B so all computational paths of M reject 1^n. This shows that the nondeterministic (linear-time) Turing machine M accepts the language L_N, and thus $L_B \in \mathbf{NP}^B$.

To show that L_B is not in $\mathbf{P}^B$, we design B as follows. Let $M_1, M_2, \ldots$ be an enumeration of all polynomial-time oracle Turing machines. We may assume for simplicity that M_i runs in time n^i. The construction proceeds in stages, where stage i constructs a part of B, which ensures that M_i^B doesn't decide L_B. We construct B by declaring that certain strings are in B and others aren't in B. Each stage determines the status of only a finite number of strings. Initially, we have no information about B. We begin with stage 1.

Stage i. So far, a finite number of strings have been declared to be in or our of A. We choose n_i greater than the length of any such string and large enough that 2^{n_i} is greater than ${n_i}^i$, the running time of M_i. We show how to extend our information about B so that M_i^B accepts 1^{n_i} whenever that string is not in L_B.

We run M_i on input 1^{n_i} and respond to its oracle queries as follows.

1. If M_i queries a string y whose status had already been determined. (This can happen because M_i, when running on input 1^{n_i}, may ask about strings y of length shorter than n_i, for which the membership of y in B had been decided from earlier stages.) In this case, we respond consistently.
2. If M_i queries a string y whose status is undetermined. (This occurs when M_i queries strings y of length n_i or greater, for which the membership of y in B has not been decided yet.) In this case, we respond NO to the query and declare y to be out of B.

We continue the simulation of M_i until it halts.

Now consider the situation from M_i's perspective. If it finds a string of length n_i in B, it should accept because it knows that 1^{n_i} is in L_B. If M_i determines that all strings of length n_i aren't in B, it should reject because it knows that 1^{n_i} is not in L_B. However, it doesn't have enough time to ask about all strings of length n_i, and we have answered NO to each of the queries it has made. Hence, when M_i halts and must decide whether to accept or reject, it doesn't have enough information to be sure that its decision is correct.

Our objective is to ensure that its decision is *not* correct. We do so by observing its decision and then extending B so that the reverse is true. Let us denote by B_i the subset of B that is constructed at stage i. At each stage i, it is easy to see that the set B_i is a finite set and contains strings of length bounded by n_i. We also have $B_{i-1} \subseteq B_i$ and $n_{i-1} < n_i$ for all i, by virtue of our construction. Now, if M_i accepts 1^{n_i}, we declare all strings of length n_i to be out of B to guarantee that 1^{n_i} is not in L_B. Formally, we do this by setting $B_i = B_{i-1}$. Otherwise, if M_i rejects 1^{n_i}, we find a string of length n_i that M_i hasn't queried and declare that string to be in B to guarantee that 1^{n_i} is in L_B.

Such a string must exist because M_i runs for ${n_i}^i$ steps, which is fewer than 2^{n_i}, the total number of strings of length n_i. Formally, let Q_i be the set of queries made by M_i, pick any string $x \in \{0,1\}^{n_i} \setminus Q_i$, then we set $B_i = B_{i-1} \cup \{x\}$. In either case, we have ensured that M_i^B doesn't decide L_B. Stage i is completed and we proceed with stage $i+1$.

After finishing all stages, we complete the construction of B by arbitrarily declaring that any string whose status remains undetermined by all stages is out of B. No polynomial-time oracle Turing machine decides L_B with oracle B, proving the theorem. □

§1.4.3 Logical independence vs relativization

If relativization seems too banal, one way to appreciate it is to try to invent new techniques, for proving inclusions or separations among complexity classes, that *fail* to relativize. It's much harder than it sounds! A partial explanation for this challenge was given by Arora, Impagliazzo, and Vazirani [ArImVa92], by drawing an analogy between the relativization barrier and *independence* results in mathematical logic.

An independence result shows that certain mathematical statements cannot be proven or disproven within a given formal system of axioms. Classic examples include the independence of Euclid's fifth postulate from the first four (which led to the development of non-Euclidean geometry), and the independence of the continuum hypothesis from Zermelo-Fraenkel set theory. Similarly, relativization results—such as the fact that $\mathbf{P} \neq \mathbf{NP}$ cannot be proven or disproven with "known techniques"—are akin to these logical independence results. However, our results are less precise, as the term "known techniques" is vague and not as formally defined as the axiomatic systems in logic.

So, can techniques like diagonalization or simulation help resolve the **P** vs **NP** question? Possibly, but any such approach must rely on a fact about Turing machines that does not hold in the presence of oracles, i.e. a *non-relativizing* fact. Even though many results in complexity relativize, there are notable exceptions, such as $\mathbf{IP} = \mathbf{PSPACE}$ and the **PCP** theorem. Yet, despite these non-relativizing results, we still don't know how to apply them to resolve the **P** vs **NP** question!

Arora, Impagliazzo, and Vazirani further expanded on this analogy by explaining that a relativizing proof is simply any proof that "knows" about complexity classes, only through axioms that assert the classes' closure properties, as well as languages that the classes do contain. For instance, the class **P** includes the empty language, and if two languages L_1 and L_2 are in **P**, then so are Boolean combinations like $\overline{L_1}$ and $L_1 \cap L_2$. Using these closure properties, one can prove results like $\mathbf{P} \neq \mathbf{EXPTIME}$. However, when it comes to more intricate statements, like $\mathbf{P} \neq \mathbf{NP}$, these closure axioms alone are not enough to prove or disprove the claim. In fact, the statement $\mathbf{P} \neq \mathbf{NP}$ can be shown to be independent of the axioms. To demonstrate this, one can construct models of axioms where the statement $\mathbf{P} \neq \mathbf{NP}$ is false. This is done by using oracles to "force in" additional languages—such as **PSPACE**-complete languages, as seen in our proof of the Baker-Gill-Solovay theorem—which the axioms don't require to be in **P** or **NP**, but also don't prohibit. The conclusion is that any proof of $\mathbf{P} \neq \mathbf{NP}$ will need to appeal to deeper properties of these classes—properties that go beyond the closure axioms and cannot be captured by relativizing techniques.

Bibliography

[**AgKaSa04**] Manindra Agrawal, Neeraj Kayal, and Nitin Saxena. PRIMES is in P. In: *Annals of mathematics*, (2004), pp. 781–793 (cited p. 21)

[**ArBa09**] Sanjeev Arora and Boaz Barak. *Computational complexity: a modern approach*. Cambridge University Press, 2009 (cited pp. 8, 12)

[**ArImVa92**] Sanjeev Arora, Russell Impagliazzo, and Umesh Vazirani. *Relativizing versus nonrelativizing techniques: the role of local checkability*. 1992 (cited p. 25)

[**Ba16**] László Babai. "Graph isomorphism in quasipolynomial time". In: *Proceedings of the forty-eighth annual ACM symposium on Theory of Computing*. 2016, pp. 684–697 (cited p. 21)

[**BaGiSo75**] Theodore Baker, John Gill, and Robert Solovay. Relativizations of the P=?NP question. In: *SIAM Journal on computing*, **4**:4 (1975), pp. 431–442 (cited p. 21)

[**Be21**] Shalev Ben-David. *NP intuitions and Ladner's theorem*. 2021. URL: https://cs.uwaterloo.ca/~s4bendav/files/CS360S21Lec24.pdf (cited p. 15)

[**Co72**] Stephen A Cook. "A hierarchy for nondeterministic time complexity". In: *Proceedings of the fourth annual ACM symposium on Theory of computing*. 1972, pp. 187–192 (cited p. 8)

[**DoFo03**] Rod Downey and Lance Fortnow. Uniformly hard languages. In: *Theoretical Computer Science*, **298**:2 (2003), pp. 303–315 (cited p. 21)

[**FoSa11**] Lance Fortnow and Rahul Santhanam. "Robust simulations and significant separations". In: *International Colloquium on Automata, Languages, and Programming*. Springer. 2011, pp. 569–580 (cited p. 12)

[**GaJo79**] Michael R Garey and David S Johnson. *Computers and intractability*. Vol. 174. freeman San Francisco, 1979 (cited p. 14)

[**Go08**] Oded Goldreich. Computational complexity: a conceptual perspective. In: (2008) (cited p. 3)

[**Ha68**] Juris Hartmanis. Computational complexity of one-tape Turing machine computations. In: *Journal of the ACM (JACM)*, **15**:2 (1968), pp. 325–339 (cited p. 7)

[**HaSt65**] Juris Hartmanis and Richard E Stearns. On the computational complexity of algorithms. In: *Transactions of the American Mathematical Society*, **117**: (1965), pp. 285–306 (cited p. 6)

[**HeSt66**] Fred C Hennie and Richard Edwin Stearns. Two-tape simulation of multitape Turing machines. In: *Journal of the ACM (JACM)*, **13**:4 (1966), pp. 533–546 (cited p. 6)

[**Kh80**] Leonid G Khachiyan. Polynomial algorithms in linear programming. In: *USSR Computational Mathematics and Mathematical Physics*, **20**:1 (1980), pp. 53–72 (cited p. 21)

[**La75**] Richard E Ladner. On the structure of polynomial time reducibility. In: *Journal of the ACM (JACM)*, **22**:1 (1975), pp. 155–171 (cited pp. 14, 21)

[**SeFiMe78**] Joel I Seiferas, Michael J Fischer, and Albert R Meyer. Separating nondeterministic time complexity classes. In: *Journal of the ACM (JACM)*, **25**:1 (1978), pp. 146–167 (cited p. 8)

[**Si13**] Michael Sipser. *Introduction to the Theory of Computation*. Cengage Learning, 2013 (cited p. 4)

[**Žá83**] Stanislav Žák. A Turing machine time hierarchy. In: *Theoretical Computer Science*, **26**:3 (1983), pp. 327–333 (cited p. 9)