Chapter 4

Diagonalization

"..the relativized **P** $=?$**NP** *question has a positive answer for some oracles and a negative answer for other oracles. We feel that this is further evidence of the difficulty of the* **P** $=?$**NP** *question."*
Baker, Gill, Solovay. [**?**]

One basic goal in complexity theory is to separate interesting complexity classes. To separate two complexity classes we need to exhibit a machine in one class that gives a different answer on some input from *every* machine in the other class. This chapter describes *diagonalization*, essentially the only general technique known for constructing such a machine. The first use of diagonalization is to prove *hierarchy theorems*, according to which giving Turing machines more computational resources (such as time, space, and non-determinism) allows them to solve a strictly larger number of problems. We will also use it to show that if **P** $\neq$ **NP** then there exist problems that are neither in **P** nor **NP**-complete.

Though diagonalization led to some of these early successes of complexity theory, researchers realized in the 1970s that diagonalization alone may not resolve **P** versus **NP** and other interesting questions; see Section 4.5. Interestingly, the limits of diagonalization are proved using diagonalization.

This last result caused diagonalization to go out of favor for many years. But some recent results (see Section 17.3 for an example) use diagonalization as a key component. Thus future complexity theorists should master this simple idea before going on to anything fancier!

Machines as strings and the universal TM. The one common tool used in all diagonalization proofs is the representation of TMs by strings, such that given a string x a *universal* TM can simulate the machine M_x represented by x with a small (i.e. at most logarithmic) overhead, see Theorems 1.6, **??** and **??**. Recall that we assume that every string x represents some machine and every machine is represented by infinitely many strings. For $i \in \mathbb{N}$, we will also use the notation M_i for the machine represented by the string that is the binary expansion of the number i.

4.1 Time Hierarchy Theorem

The Time Hierarchy Theorem shows that allowing Turing Machines more computation time strictly increases the class of languages that they can decide. Recall that a function $f : \mathbb{N} \to \mathbb{N}$ is a *time-constructible* function if there is a Turing machine that, given the input 1^n, writes down $1^{f(n)}$ on its tape in $O(f(n))$ time. Usual functions like $n \log n$ or n^2 satisfy this property, and we will restrict attention to running times that are time-constructible.

Theorem 4.1
If f, g are time-constructible functions satisfying $f(n) \log f(n) = o(g(n))$, then

$$\mathbf{DTIME}(f(n)) \subsetneq \mathbf{DTIME}(g(n)) \tag{1}$$

Proof: To showcase the essential idea of the proof of Theorem 4.1, we prove the simpler statement $\mathbf{DTIME}(n) \subsetneq \mathbf{DTIME}(n^{1.5})$.

Consider the following Turing Machine D: "*On input x, run for $|x|^{1.4}$ steps the Universal TM $\mathcal{U}$ of Theorem 1.6 to simulate the execution of M_x on x. If M_x outputs an answer in this time, namely, $M_x(x) \in \{0,1\}$ then* output the opposite answer *(i.e., output $1 - M_x(x)$). Else output 0.*" Here M_x is the machine represented by the string x.

By definition, D halts within $n^{1.4}$ steps and hence the language L decided by D is in $\mathbf{DTIME}(n^{1.5})$. We claim that $L \notin \mathbf{DTIME}(n)$.

For contradiction's sake assume that some TM M decides L but runs in time cn on inputs of size n. Then every $x \in \{0,1\}^*$, $M(x) = D(x)$.

The time to simulate M by the universal Turing machine $\mathcal{U}$ on every input x is at most $c'c|x|\log|x|$ for some constant c' (depending on the alphabet size and number of tapes and states of M, but independent of $|x|$). There exists a number n_0 such that for every $n \geq n_0$, $n^{1.4} > c'cn \log n$. Let x be a string representing the machine M of length at least n_0 (there exists such a string since M is represented by infinitely many strings). Then, $D(x)$

will obtain the output $M(x)$ within $|x|^{1.4}$ steps, but by definition of D, we have $D(x) = 1 - M(x) \neq M(x)$. Thus we have derived a contradiction. ■

Figure 4.1 shows why this method is called "diagonalization".

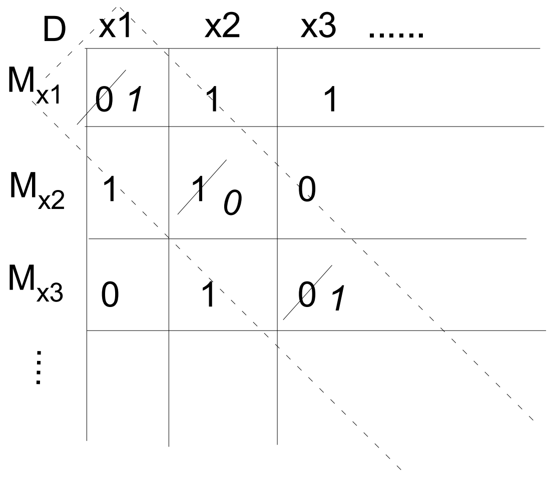

Figure 4.1: If we order all the strings in $\{0,1\}^*$ as $x_1, x_2, \ldots$ and have a table that contains in its $\langle i,j \rangle th$ location the value of the machine represented by x_i on the input x_i, then on large enough inputs, the diagonalizer D from the proof of Theorem 4.1 computes the function that is the negation of this table's diagonal.

4.2 Space Hierarchy Theorem

The space hierarchy theorem is completely analogous to the time hierarchy theorem. One restricts attention to *space-constructible* functions, which are functions $f : \mathbb{N} \to \mathbb{N}$ for which there is a machine that, given any n-bit input, constructs $f(n)$ in space $O(f(n))$. The proof of the next theorem is completely analogous to that of Theorem 4.1. (The theorem does not have the $\log f(n)$ factor because the universal machine for space-bounded computation incurs only a constant factor overhead in space; see Theorem **??**.)

THEOREM 4.2
If f, g are space-constructible functions satisfying $f(n) = o(g(n))$, then

$$\mathbf{SPACE}(f(n)) \subsetneq \mathbf{SPACE}(g(n)) \tag{2}$$

4.3 Nondeterministic Time Hierarchy Theorem

The following is the hierarchy theorem for *non-deterministic* Turing machines.

Theorem 4.3
If f, g are time constructible functions satisfying $f(n+1) = o(g(n))$, then

$$\mathbf{NTIME}(f(n)) \subsetneq \mathbf{NTIME}(g(n)) \tag{3}$$

Proof: Again, we just showcase the main idea by proving $\mathbf{NTIME}(n) \subsetneq \mathbf{NTIME}(n^{1.5})$. The technique from the previous section does not directly apply, since it has to determine the answer of a TM in order to flip it. To determine the answer of a nondeterminisitic that runs in $O(n)$ time, we may need to examine as many as $2^{\Omega(n)}$ possible strings of non-deterministic choices. So it is unclear that how the "diagonalizer" machine can determine in $O(n^{1.5})$ (or even $O(n^{100})$) time how to flip this answer. Instead we introduce a technique called *lazy* diagonalization, which is only guaranteed to flip the answer on some input in a fairly large range.

For every $i \in \mathbb{N}$ we denote by M_i the non-deterministic TM represented by i's binary expansion according to the universal NDTM $\mathcal{NU}$ (see Theorem **??**). We define the function $f : \mathbb{N} \to \mathbb{N}$ as follows: $f(1) = 2$ and $f(i+1) = 2^{f(i)^{1.2}}$. Note that given n, we can can easily find in $O(n^{1.5})$ time the number i such that n is sandwiched between $f(i)$ and $f(i+1)$. Our diagonalizing machine D will try to flip the answer of M_i on *some* input in the set $\{1^n : f(i) < n \le f(i+1)\}$. It is defined as follows:

"*On input x, if $x \notin 1^*$, reject. If $x = 1^n$, then compute i such that $f(i) < n \le f(i+1)$ and*

1. *If $f(i) < n < f(i+1)$ then simulate M_i on input 1^{n+1} using nondeterminism in $n^{1.1}$ time and output its answer. (If the simulation takes more than that then halt and accept.)*

2. *If $n = f(i+1)$, accept 1^n iff M_i rejects $1^{f(i)+1}$ in $(f(i)+1)^{1.5}$ time."*

Note that Part 2 requires going through all possible $\exp((f(i)+1)^{1.1})$ branches of M_i on input $1^{f(i)+1}$, but that is fine since the input size $f(i+1)$ is $2^{f(i)^{1.2}}$. We conclude that NDTM D runs in $O(n^{1.5})$ time. Let L be the language decided by D. We claim that $L \notin \mathbf{NTIME}(n)$.

Indeed, suppose for the sake of contradiction that L is decided by an NDTM M running in cn steps (for some constant c). Since each NDTM is represented by infinitely many strings, we can find i large enough such that

$M = M_i$ and on inputs of length $n \geq f(i)$, M_i can be simulated in less than $n^{1.1}$ steps. Thus the two steps in the description of D ensure respectively that

$$\text{If } f(i) < n < f(i+1), \quad \text{then } D(1^n) = M_i(1^{n+1}) \tag{4}$$
$$D(1^{f(i+1)}) \neq M_i(1^{f(i)+1}) \tag{5}$$

see Figure 4.2.

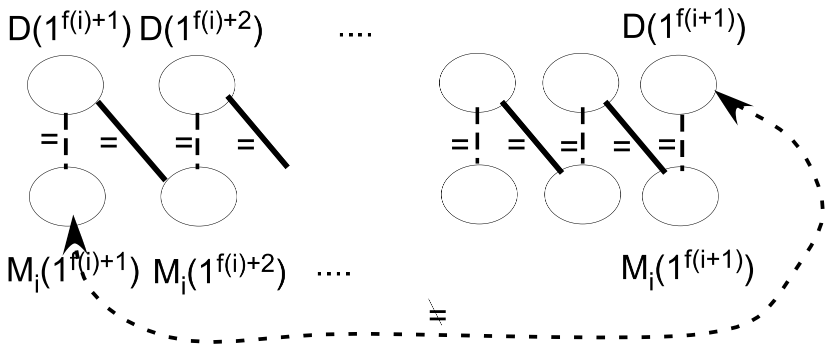

Figure 4.2: The values of D and M_i on inputs 1^n for $n \in (f(i), f(i+1)]$. Full lines denote equality by the design of D, dashed lines denote equality by the assumption that $D(x) = M_i(x)$ for every x, and the dashed arrow denotes inequality by the design of D. Note that together all these relations lead to contradiction.

By our assumption M_i and D agree on all inputs 1^n for $n \in (f(i), f(i+1)]$. Together with (4), this implies that $D(1^{f(i+1)}) = M_i(1^{f(i)+1})$, contradicting(5). ■

4.4 Ladner's Theorem: Existence of NP-intermediate problems.

One of the striking aspects of **NP**-completeness is the surprisingly large number of **NP**-problems –including some that were studied for many decades— that turned out to be **NP**-complete. This phenomenon suggests a bold conjecture: every problem in **NP** is either in **P** or **NP** complete. We show that if $\mathbf{P} \neq \mathbf{NP}$ then this is false —there is a language $L \in \mathbf{NP} \setminus \mathbf{P}$ that is not **NP**-complete. (If $\mathbf{P} = \mathbf{NP}$ then the conjecture is trivially true but uninteresting.) The rest of this section proves this.

THEOREM 4.4 (LADNER'S THEOREM [**?**])
Suppose that $\mathbf{P} \neq \mathbf{NP}$*. Then there exists a language* $L \in \mathbf{NP} \setminus \mathbf{P}$ *that is not* **NP***-complete.*

PROOF: If $\mathbf{P} \neq \mathbf{NP}$ then we know at least one language in $\mathbf{NP} \setminus \mathbf{P}$: namely, the **NP**-complete language SAT. Consider the language SAT_H of length n satisfiable formulae that are padded with $n^{H(n)}$ 1's for some polynomial-time computable function $H : \mathbb{N} \to \mathbb{N}$ (i.e., $\mathsf{SAT}_H = \left\{\psi 01^{n^{H(n)}} : \psi \in \mathsf{SAT} \text{ and } n = |\psi|\right\}$). Consider two possibilities:

(a) *$H(n)$ is at most some constant c for every n.* In this case SAT_H is simply SAT with a polynomial amount of "padding." Thus, SAT_H is also **NP**-complete and is not in **P** if $\mathbf{P} \neq \mathbf{NP}$.

(b) *$H(n)$ tends to infinity with n*, and thus the padding is of superpolynomial size. In this case, we claim that SAT_H cannot be **NP**-complete. Indeed, if there is a $O(n^i)$-time reduction f from SAT to SAT_H then such a reduction reduces the satisfiability of SAT instances of length n to instances of SAT_H of length $O(n^i)$, which must have the form $\psi 01^{|\psi|^{H(|\psi|)}}$, where $|\psi| + |\psi|^{H(|\psi|)} = O(n^i)$, and hence $|\psi| = o(n)$. In other words, we have a polynomial-time reduction from SAT instances of length n to SAT instances of length $o(n)$, which implies SAT can be solved in polynomial time. (The algorithm consists of applying the reduction again and again, reducing the size of the instances each time until the instance is of size $O(1)$ and can be solved in $O(1)$ time by brute force) This is a contradiction to the assumption $\mathbf{P} \neq \mathbf{NP}$.

The proof of the Theorem uses a language SAT_H for a function H that in some senses combines the two cases above. This function tends to infinity with n, so that SAT_H is not **NP**-complete as in Case (b), but grows slowly enough to assure $\mathsf{SAT}_H \notin \mathbf{P}$ as in Case (a). Function H is defined as follows:

> $H(n)$ is the smallest number $i < \log\log n$ such that for every $x \in \{0,1\}^*$ with $|x| \leq \log n$,
>
> M_i halts on x within $i|x|^i$ steps and M_i outputs 1 iff $x \in \mathsf{SAT}_H$
>
> where M_i is the machine represented by the binary expansion of i according to the representation scheme of the universal Turing machine $\mathcal{U}$ of Theorem 1.6. If there is no such i then we let $H(n) = \log\log n$.

Notice, this is implicitly a recursive definition since the definition of H depends on SAT_H, but a moment's thought shows that H is well-defined since $H(n)$ determines membership in SAT_H of strings whose length is greater than n, and the definition of $H(n)$ only relies upon checking the status of strings of length at most $\log n$.

There is a trivial algorithm to compute $H(n)$ in $O(n^3)$ time. After all, we only need to **(1)** compute $H(k)$ for every $k \leq \log n$, **(2)** simulate at most $\log\log n$ machines for every input of length at most $\log n$ for $\log\log n(\log n)^{\log\log n} = o(n)$ steps, and **(3)** compute SAT on all the inputs of length at most $\log n$.

Now we have the following two claims.

CLAIM 1: *SAT_H is not in* $\mathbf{P}$*.* Suppose, for the sake of contradiction, that there is a machine M solving SAT_H in at most cn^c steps. Since M is represented by infinitely many strings, there is a number $i > c$ such that $M = M_i$. By the definition of $H(n)$ this implies that for $n > 2^{2^i}$, $H(n) \leq i$. But this means that for all sufficiently large input lengths, SAT_H is simply the language SAT padded with a polynomial (i.e., n^i) number of 1's, and so cannot be in $\mathbf{P}$ unless $\mathbf{P} = \mathbf{NP}$.

CLAIM 2: *SAT_H is not* $\mathbf{NP}$*-complete.* As in Case (b), it suffices to show that $H(n)$ tends to infinity with n. We prove the equivalent statement that for every integer i, there are only finitely many n's such that $H(n) = i$: since $\mathsf{SAT}_H \notin \mathbf{P}$, for each i we know that there is an input x such that given $i|x|^i$ time, M_i gives the incorrect answer to whether or not $x \in \mathsf{SAT}_H$. Then the definition of H ensures that for every $n > 2^{|x|}$, $H(x) \neq i$.

■

REMARK 4.5
We do not know of a natural decision problem that, assuming $\mathbf{NP} \neq \mathbf{P}$, is proven to be in $\mathbf{NP} \setminus \mathbf{P}$ but not $\mathbf{NP}$-complete, and there are remarkably few candidates for such languages. However, there are a few fascinating examples for languages not known to be either in $\mathbf{P}$ nor $\mathbf{NP}$-complete. Two such examples are the *Factoring* and *Graph isomorphism* languages (see Example 2.3). No polynomial-time algorithm is currently known for these languages, and there is some evidence that they are not $\mathbf{NP}$ complete (see Chapter 9).

4.5 Oracle machines and the limits of diagonalization?

Quantifying the limits of "diagonalization" is not easy. Certainly, the diagonalization in Sections 4.3 and 4.4 seems more clever than the one in Section 4.1 or the one that proves the undecidability of the halting problem.

For concreteness, let us say that "diagonalization" is any technique that relies upon the following properties of Turing machines

1. The existence of an effective representation of Turing machines by strings.

2. The ability of one TM to simulate any another without much overhead in running time or space.

Any argument that only uses these facts is treating machines as *black-boxes*: the machine's internal workings do not matter. We show a general way to define a variant of Turing Machines called *oracle Turing Machines* that still satisfy the above two properties. However, one way of defining the variants results in TMs for which $\mathbf{P} = \mathbf{NP}$, whereas the other way results in TMs for which $\mathbf{P} \neq \mathbf{NP}$. We conclude that to resolve $\mathbf{P}$ versus $\mathbf{NP}$ we need to use some other property besides the above two.

Oracle machines will be used elsewhere in this book in other contexts. These are machines that are given access to an "oracle" that can magically solve the decision problem for some language $O \subseteq \{0,1\}^*$. The machine has a special *oracle tape* on which it can write a string $q \in \{0,1\}^*$ on a and in one step gets an answer to a query of the form "Is q in O?". This can be repeated arbitrarily often with different queries. If O is a difficult language that cannot be decided in polynomial time then this oracle gives an added power to the TM.

Definition 4.6 (Oracle Turing Machines)
An *oracle* Turing machine is a TM M that has a special read/write tape we call M's *oracle tape* and three special states $q_{\mathsf{query}}, q_{\mathsf{yes}}, q_{\mathsf{no}}$. To execute M, we specify in addition to the input a language $O \subseteq \{0,1\}^*$ that is used as the *oracle* for M. Whenever during the execution M enters the state q_{query}, the machine moves into the state q_{yes} if $q \in O$ and q_{no} if $q \notin O$, where q denotes the contents of the special oracle tape. Note that, regardless of the choice of O, a membership query to O counts only as a single computational step. If M is an oracle machine, $O \subseteq \{0,1\}^*$ a language, and $x \in \{0,1\}^*$, then we denote the output of M on input x and with oracle O by $M^O(x)$.

Nondeterministic oracle TMs are defined similarly.

DEFINITION 4.7
For every $O \subseteq \{0,1\}^*$, $\mathbf{P}^O$ is the set of languages decided by a polynomial-time deterministic TM with oracle access to O and $\mathbf{NP}^O$ is the set of languages decided by a polynomial-time non-deterministic TM with oracle access to O.

To illustrate these definitions we show a few simple claims.

CLAIM 4.8

1. *Let* $\overline{\mathsf{SAT}}$ *denote the language of* unsatisfiable *formulae. Then* $\overline{\mathsf{SAT}} \in \mathbf{P}^{\mathsf{SAT}}$.

2. *Let* $O \in \mathbf{P}$*. Then* $\mathbf{P}^O = \mathbf{P}$.

3. *Let* TQBF *be the* **PSPACE***-complete language of true quantified Boolean formulae (see Section 3.3). Then* $\mathbf{P}^{\mathsf{TQBF}} = \mathbf{NP}^{\mathsf{TQBF}} = \mathbf{PSPACE}$.

PROOF:

1. Given oracle access to SAT, to decide whether a formula φ is in $\overline{\mathsf{SAT}}$, the machine asks the oracle if $\varphi \in \mathsf{SAT}$, and then gives the opposite answer as its output.

2. Allowing an oracle can only help compute more languages and so $\mathbf{P} \subseteq \mathbf{P}^O$. If $O \in \mathbf{P}$ then it is redundant as an oracle, since we can transform any polynomial-time oracle TM using O into a standard (no oracle) by simply replacing each oracle call with the computation of O. Thus $\mathbf{P}^O \subseteq \mathbf{P}$.

3. Because TQBF is **PSPACE**-complete, we can decide every language in **PSPACE** using one oracle call to it, and hence $\mathbf{PSPACE} \subseteq \mathbf{P}^{\mathsf{TQBF}}$. Note also that clearly $\mathbf{P}^{\mathsf{TQBF}} \subseteq \mathbf{NP}^{\mathsf{TQBF}}$. If M is a non-deterministic polynomial-time oracle TM, we can simulate its execution with a TQBF oracle in polynomial space: it only requires polynomial space to enumerate all of M's non-deterministic choices and to solve the TQBF oracle queries. Thus, $\mathbf{PSPACE} \subseteq \mathbf{P}^{\mathsf{TQBF}} \subseteq \mathbf{NP}^{\mathsf{TQBF}} \subseteq \mathbf{PSPACE}$.

■

The key fact to note about oracle TMs is the following: *Regardless of what oracle O is, the set of all oracle TM's with access to oracle O satisfy Properties 1 and 2 above.* The reason is that we can represent TMs with oracle O as strings, and we have a universal TM $\mathcal{OU}$ that, using access to O, can simulate every such machine with logarithmic overhead, just as Theorem 1.6 shows for non-oracle machines. Indeed, we can prove this in exactly the same way of Theorem 1.6, except that whenever in the simulation M makes an oracle query, $\mathcal{OU}$ forwards the query to its own oracle.

Thus any result about TMs or complexity classes that uses only Properties 1 and 2 above also holds for the set of all TMs with oracle O. Such results are called *relativizing* results.

All of the results on universal Turing machines and the diagonalizations results in this chapter are of this type.

The next theorem implies that whichever of $\mathbf{P} = \mathbf{NP}$ or $\mathbf{P} \neq \mathbf{NP}$ is true, it cannot be a relativizing result.

Theorem 4.9 (Baker, Gill, Solovay [**?**])
There exist oracles A, B such that $\mathbf{P}^A = \mathbf{NP}^A$ and $\mathbf{P}^B \neq \mathbf{NP}^B$.

Proof: As seen in Claim 4.8, we can use $A = \mathsf{TQBF}$. Now we construct B. For any language B, let U_B be the unary language

$$U_B = \{1^n : \text{some string of length } n \text{ is in } B\}.$$

For every oracle B, the language U_B is clearly in $\mathbf{NP}^B$, since a non-deterministic TM can make a non-deterministic guess for the string $x \in \{0,1\}^n$ such that $x \in B$. Below we construct an oracle B such that $U_B \notin \mathbf{P}^B$, implying that $\mathbf{P}^B \neq \mathbf{NP}^B$.

Construction of B: For every i, we let M_i be the oracle TM represented by the binary expansion of i. We construct B in stages, where stage i ensures that M_i^B does not decide U_B in $2^n/10$ time. Initially we let B be empty, and gradually add strings to it. Each stage determines the status (i.e., whether or not they will ultimately be in B) of a finite number of strings.

Stage i: So far, we have declared for a finite number of strings whether or not they are in B. Choose n large enough so that it exceeds the length of any such string, and run M_i on input 1^n for $2^n/10$ steps. Whenever it queries the oracle about strings whose status has been determined, we answer consistently. When it queries strings whose status is undetermined, we declare that the string is not in B. Note that until this point, we have not declared that B has any string of length n. Now we make sure that if M_i halts within $2^n/10$ steps then its answer on 1^n is incorrect. If M_i accepts, we declare that all strings of length n are not in B, thus ensuring $1^n \notin B_u$. If M_i rejects, we pick a string of length n that it has not queried (such a string exists because M_i made at most $2^n/10$ queries) and declare that it is in B, thus ensuring $1^n \in B_u$. In either case, the answer of M_i is incorrect. Our construction ensures that U_B is not in $\mathbf{P}^B$ (and in fact not in $\mathbf{DTIME}^B(f(n))$ for every $f(n) = o(2^n)$). ■

Let us now answer our original question: Can diagonalization or any simulation method resolve **P** vs **NP**? Answer: Possibly, but it has to use some fact about TMs that does not hold in presence of oracles. Such facts are termed *nonrelativizing* and we will later see examples of such facts. However, a simple one was already encountered in Chapter **??**: the Cook-Levin theorem! It is not true for a general oracle A that every language $L \in \mathbf{NP}^A$ is polynomial-time reducible to 3SAT (see Exercise 6). Note however that nonrelativizing facts are necessary, not sufficient. It is an open question how to use known nonrelativizing facts in resolving **P** vs **NP** (and many interesting complexity theoretic conjectures).

Whenever we prove a complexity-theoretic fact, it is useful to check whether or not it can be proved using relativizing techniques. The reader should check that Savitch's theorem (Corollary **??**) and Theorem 3.18 do relativize.

Later in the book we see other attempts to separate complexity classes, and we will also try to quantify —using complexity theory itself!—why they do not work for the **P** versus **NP** question.

WHAT HAVE WE LEARNED?

- Diagonalization uses the representation of Turing machines as strings to separate complexity classes.
- We can use it to show that giving a TM more of the same type of resource (time, non-determinism, space) allows it to solve more problems, and to show that, assuming $\mathbf{NP} \neq \mathbf{P}$, **NP** has problems neither in **P** nor **NP**-complete.
- Results proven solely using diagonalization *relativize* in the sense that they hold also for TM's with oracle access to O, for every oracle $O \subseteq \{0,1\}^*$. We can use this to show the limitations of such methods. In particular, relativizing methods alone cannot resolve the **P** vs. **NP** question.

Chapter notes and history

Georg Cantor invented diagonalization in the 19th century to show that the set of real numbers is uncountable. Kurt Gödel used a similar technique in his proof of the *Incompleteness Theorem.* Computer science undergraduates often encounter diagonalization when they are taught the undecidabilty of the *Halting Problem.*

The time hierarchy theorem is from Hartmanis and Stearns' pioneering paper [**?**]. The space hierarchy theorem is from Stearns, Hartmanis, and Lewis [**?**]. The nondeterministic time hierarchy theorem is from Cook [**?**], though the simple proof given here is essentially from [**?**]. A similar proof works for other complexity classes such as the (levels of the) polynomial hierarchy discussed in the next chapter. Ladner's theorem is from [**?**] but the proof here is due to an unpublished manuscript by Impagliazzo. The notion of relativizations of the **P** versus **NP** question is from Baker, Gill, and Solovay [**?**], though the authors of that paper note that other researchers independently discovered some of their ideas. The notion of relativization is related to similar ideas in logic (such as *independence results*) and recursive function theory.

The notion of oracle Turing machines can be used to study interrelationships of complexity classes. In fact, Cook [**?**] defined **NP**-completeness using oracle machines. A subfield of complexity theory called *structural complexity* has carried out a detailed study of oracle machines and classes defined using them; see [].

Whether or not the Cook-Levin theorem is a nonrelativizing fact depends upon how you formalize the question. There is a way to allow the 3SAT instance to "query" the oracle, and then the Cook-Levin theorem does relativize. However, it seems safe to say that any result that uses the locality of computation is looking at the internal workings of the machine and hence is potentially nonrelativizing.

The term *superiority* introduced in the exercises does not appear in the literature but the concept does. In particular, ??? have shown the limitations of relativizing techniques in resolving certain similar open questions.

Exercises

§1 Show that the following language is undecidable:

$$\left\{ \llcorner M \lrcorner : \ M \text{ is a machine that runs in } 100n^2 + 200 \text{ time} \right\}.$$

§2 Show that $\mathbf{SPACE}(n) \neq \mathbf{NP}$. (Note that we do not know if either class is contained in the other.)

§3 Show that there is a language $B \in \mathbf{EXP}$ such that $\mathbf{NP}^B \neq \mathbf{P}^B$.

§4 Say that a class C_1 is *superior to* a class C_2 if there is a machine M_1 in class C_1 such that for every machine M_2 in class C_2 and every large enough n, there is an input of size between n and n^2 on which M_1 and M_2 answer differently.

(a) Is $\mathbf{DTIME}(n^{1.1})$ superior to $\mathbf{DTIME}(n)$?

(b) Is $\mathbf{NTIME}(n^{1.1})$ superior to $\mathbf{NTIME}(n)$?

§5 Show that there exists a function that is not time-constructible.

§6 Show that there is an oracle A and a language $L \in \mathbf{NP}^A$ such that L is not polynomial-time reducible to 3SAT even when the machine computing the reduction is allowed access to A.

§7 Suppose we pick a random language B, by deciding for each string independently and with probability $1/2$ whether or not it is in B. Show that with high probability $\mathbf{P}^B \neq \mathbf{NP}^B$. (To give an answer that is formally correct you may need to know elementary measure theory.)