Chapter 1

The computational model
—and why it doesn’t matter

“The subject of my talk is perhaps most directly indicated by simply asking two questions: first, is it harder to multiply than to add? and second, why?...I (would like to) show that there is no algorithm for multiplication computationally as simple as that for addition, and this proves something of a stumbling block.”

Alan Cobham, 1964 [Cob64]

The invention of a formal model for describing algorithms —shortly followed by its implementation in the form of the electronic computer in the 1940s— is one of the chief scientific achievements of the 20th century. In fact, more than one computational model was conceived, but in the end all were proven equivalent (i.e., each could simulate the others). Today, of course, we take this equivalence for granted, since we have seen a plethora of programming languages and know that compilers can translate programs from one language into the other.

Early researchers took a while to realize this, however, but having done that they could focus on computation in the abstract and try to understand its power. Research from the 1930s through the 1950s developed a theory of which algorithmic problems are computable. Many interesting algorithmic tasks were found to be uncomputable or undecidable: no computer can solve them without going into infinite loops (i.e., never halting) on certain inputs. This beautiful theory is usually taught in undergraduate courses and sometimes in logic departments. Though it still has many interesting open problems, it will not concern us here.
Rather, we will be interested in issues of \textit{computational efficiency}, which, after the introduction of computers into scientific fields in the 1950s have taken on increased importance. Consider the problem of multiplying two \(n\)-bit numbers, a basic task in scientific computing. Since the early 1960s it is known that the trivial algorithm —familiar from elementary school— is less efficient than a new, less intuitive algorithm based upon the Fast Fourier Transform (FFT). Many other such examples of counterintuitive yet efficient algorithms are known, and newer ones are discovered every year as part of a broad research program in algorithms design. Since the 1950s, interest also grew in a large class of optimization problems for which no method was known apart from brute-force search in a large space of possible solution. The effort to understand these problems resulted in the theory of \textit{NP-completeness} in the 1970s.

The field of \textit{computational complexity} concerns itself with these and related investigations. Broadly, it studies the following type of question: how efficiently can we solve a specific computational problem on a given computational model? Of course, the model ultimately of interest is the Turing machine (TM), or equivalently, any modern programming language such as C or Java. However, while trying to understand complexity issues arising in the study of the Turing machine, we often gain interesting insight by considering modifications of the basic Turing machine —nondeterministic, alternating and probabilistic TMs, circuits, quantum TMs etc.— as well as totally different computational models—communication games, decision trees, algebraic computation trees etc. Many beautiful results of complexity theory concern such models and their interrelationships.

\subsection{1.1 Encodings and Languages: Some conventions}

Throughout, we will assume that the input and output is a finite string of bits (i.e., a member of \(\{0,1\}^*\)). Note that simple encodings can be used to represent a general mathematical object —integer, pairs of integers, graph, vector, matrix, etc.— as a string of bits. To give a simple example, nonnegative integers can be encoded by representing \(n\) using \(1\left[\frac{n}{2}\right]\) (where \(\left[\frac{n}{2}\right]\) is the binary representation of \(n\)); it is easily checked that this is a one-to-one mapping from nonnegative integers to \(\{0,1\}^*\). Similarly, pairs of integers can be encoded by representing \((n,m)\) by \(1(3 \times [n]_2)010(3 \times [m]_2)\) where \(3 \times s\) is our (clumsy) notation for the bit string obtained from string \(s\) by replicating each bit thrice. From now on we will not explicitly deal with such low level issues of representation, and will use \(<x>\) to denote some
1.2. THE TURING MACHINE

Canonical (and unspecified) binary representation of the object $x$.

Another frequent practice in complexity theory is to recast computational problems as languages. A language is a subset (potentially infinite) of $\{0,1\}^*$. As example, we recall the famous FACTORING problem, where the input is an integer $n$ and the output is any pair of nontrivial factors of $n$ (i.e., $x, y < n$ such that $n = x \cdot y$) if they exist.

The natural languages corresponding to this problem are

$$\text{PRIMES} = \{ < p > \in \mathbb{Z} : p \text{ is a prime integer} \} \quad (1.1)$$

$$\text{COMPOSITES} = \{ < n > : n \text{ is a composite integer} \} \quad (1.2)$$

Notice that this translation into languages is better suited for some contexts than others. In case of NP-complete problems it works great, but in other contexts it can sometimes lose important aspects of the original problem (e.g., membership in PRIMES and COMPOSITES can be decided in polynomial time, whereas we do not know if FACTORING can be solved in polynomial time; see Chapter 2). We will point out such issues wherever they arise.

Finally, another frequent convention concerns the complement of a language. We will think of PRIMES and COMPOSITES as complements of each other. Technically, this is incorrect, since their union is not $\{0,1\}^*$ but the set of bitstrings that are valid encodings of nonnegative integers. However, this fine point is inconsequential in a complexity setting since bit strings that are not valid encodings of integers are easy to recognize in all computational models we will study.

1.2 The Turing machine

We briefly recall the definition of the Turing machine (TM); for a detailed discussion see an undergraduate text.

A $k$-tape TM consists of a finite control and $k$ tapes each furnished with a read-write head. The tape is an infinite line of cells, each of which holds a symbol. (The symbols come belong to a finite alphabet.) At each step the machine reads the symbols in the cells directly under the $k$ read-write heads, replaces these symbols with new symbols (it has the option of not changing the tape by writing down the old symbols again), makes a transition to a new state within its finite control, and moves each head one cell to the left or to the right.

Formally, the machine is described by a function $\delta : Q \times \Gamma^k \rightarrow Q \times \Gamma^k \times \{L, R\}^k$. where $\Gamma$ is the tape alphabet, $Q$ is the finite set of states.
that the machine is allowed to be in. If the machine is in state \( q \in Q \) and \((\sigma_1, \sigma_2, \ldots, \sigma_k)\) are the symbols currently being read in the \( k \) tapes, and \( \delta(q, (\sigma_1, \ldots, \sigma_k)) = (q', (\sigma'_1, \ldots, \sigma'_k), z) \) where \( z \in \{L, R\}^k \) then at the next step the \( \sigma \) symbols will be replaced by the \( \sigma' \) symbols, the machine will be in state \( q' \), and the \( k \) heads will move left/right (i.e., L/R) as given by \( z \).

Figure 1.1: This TM has a single tape and the tape alphabet \( \Gamma \) is \( \{0, 1\} \). Each state has two outgoing transitions. Suppose the machine is in state \( q_1 \). If the symbol under the tape head is 0 then it will be replaced by 1 and the machine will move one cell to the right. If the symbol is 1 it will be replaced by 1 (in other words, will not be changed) and the machine will move one cell to the left.

All cells are initialized to a special blank symbol, and a finite non-blank string ("the input") is placed on the first tape. All heads start at the left ends of the tapes and the machine is in a special state \( q_{\text{start}} \). This is the start configuration. Each step of the computation is performed by applying the function \( \delta \) as described above. There are two other special states in \( Q \) besides \( q_{\text{start}} \), called \( q_{\text{acc}} \) and \( q_{\text{halt}} \). Once the machine is in either of these states, the transition function \( \delta \) does not allow it to further modify the tape or change states. Clearly, if the machine enters either of these two states, it has halted. In complexity theory we are only interested in machines that halt for every input in a finite number of steps.

A configuration for the TM consists of a string of the type \( \alpha q \beta \) where \( \alpha, \beta \in (\Gamma^*)^k \) and \( q \in Q \). This configuration is meant to show that currently the machine’s internal state is \( q \), and the strings to the left of the heads on the \( k \) tapes are \( \alpha_1, \alpha_2, \ldots, \alpha_k \in \Gamma^* \) respectively whereas the nonblank strings to the right of the heads are \( \beta_1, \beta_2, \ldots, \beta_k \in \Gamma^* \).

**Definition 1.1 (Acceptance of an input)** An input \( x \) is accepted by the above TM iff there is a finite sequence of configurations \( C_0, C_1, C_2, \ldots, C_T \) such that:

1. \( C_0 = C_{\text{start}} \) is the start configuration corresponding to input \( x \).
2. Each $C_i$ yields $C_{i+1}$ in one step.

3. $C_T$ is an accept configuration, of the form $\alpha q_{\text{acc}} \beta$. We assume without loss of generality that $C_T$ is the unique configuration $C_{\text{acc}} = \epsilon q_{\text{acc}} \epsilon$ (namely, $\alpha, \beta$ are empty strings) by modifying $\delta$ to ensure that the machine before entering $q_{\text{acc}}$ erases all the tapes and moves all heads all the way to the left.

The set of inputs that are accepted by the TM is called its language.

We will assume that the input symbols come from the alphabet $\{0, 1\}$. Thus a language is a set of strings over $\{0, 1\}$.

### 1.2.1 Nondeterministic TMs

The TM model defined above is also called the deterministic TM, since it evolves in a unique way once we fix the input.

A nondeterministic machine (NDTM) is defined similarly to a deterministic machine, except the transition $\delta$ gives two possible alternatives for the next step instead of a unique one. (In general, one could replace “two” by “a finite number” but two is general enough.)

![Diagram of a NDTM](image)

Figure 1.2: This NDTM has one tape and the alphabet is $\{0, 1\}$. When the machine is in state $q_1$ and reading symbol $b \in \{0, 1\}$, it can take one of two possible outgoing transitions.

Thus for each configuration $C$ there are two possible next configurations $C'$ and $C''$. The definition of when a machine “accepts” an input is unchanged from Definition 1.1.

We emphasize that if the machine can run for $T$ steps, then the number of possible configurations it could be in at the end could be $2^T$. By examining Definition 1.1 we see that the machine is said to “accept” the input if any single one of these configurations is $C_{\text{acc}}$. We can think of these configurations as the leaves of a binary tree whose root is $C_{\text{start}}$. Each path from the root to a leaf is called a branch.
Definition 1.2 (Running time, space) We say that the machine runs in $f(n)$ time if on every input of size $n$, the machine enters $q_{\text{acc}}$ or $q_{\text{halt}}$ in at most $f(n)$ steps. (If the machine is nondeterministic then every branch of the machine must do this in at most $f(n)$ steps.)

We say that the machine runs in $g(n)$ space if the number of cells used by the computation on inputs of size $n$ is at most $g(n)$. (For nondeterministic machines, again, this is true for every branch.)

Remark 1.1 All NDTMs we are interested in always halt, that is to say, every branch enters the $q_{\text{acc}}$ or $q_{\text{halt}}$ state in a finite number of steps. The NDTM's language is the set of inputs for which at least one branch ends up in $q_{\text{acc}}$.

It is worthwhile to clarify something that often confuses students. The complement of the language is the set of inputs for which all branches end up in $q_{\text{halt}}$. It is an important research question whether or not this complement can itself be the language of a nondeterministic machine with a similar running time. (See our discussion of coNP in Section 2.2.)

Many texts use the symbol $q_{\text{reject}}$ instead of $q_{\text{halt}}$, which leads students to mistakenly think that switching the identity of these two states gives an NDTM whose language is the complement of the language of the first machine. This is incorrect in general.

1.3 Turing machine: a universal computational model

The Turing machine model may at first glance look too simple to be useful, but, as indicated earlier, it can simulate any general-purpose programming language such as C or Fortran. We outline a simple proof why this is possible, using facts familiar to most students who grew up with computers.

It is easy to come up with Turing machines that do simple arithmetic: (i) given a string of 1’s, count the number of ones in it and write them in binary on the work tape (unary to binary conversion) and also to do the reverse translation. (ii) given numbers $[n]_2$ and $[m]_2$ in binary notation, calculate $[m + n]_2$, $[m \cdot n]_2$, $[m^n]_2$.

Using these basic primitives — and its infinite tape — a Turing machine can simulate the machine language of any modern digital computer, be it PC, Mac or something else. Recall that machine language has commands such as ADD R1 R2 R3 which may mean something like “If $i_1, i_2$ are the positive integers stored in registers $R_1, R_2, R_3$ then store $i_1 + i_2$ in register $R_3$.” All such commands are easy to simulate on a TM. The Turing machine uses its tape to simulate the memory of the computer (including any registers,
1.3. TURING MACHINE: A UNIVERSAL COMPUTATIONAL MODEL

stacks etc.). Unary-to-binary translation is useful for accessing, given \([n]_2\), the \(n\)th location in memory. Finally, recall that high-level programming languages such as C or JAVA can be translated —using a program called the compiler— into machine language. By simulating this machine language description, the Turing machine can simulate C and JAVA. This completes our (admittedly hand-wavy) proof of why the Turing model can simulate any programming language.

Finally, we mention a simple but crucial phenomenon: the existence of a universal Turing Machine.

**Definition 1.3 (Universal Turing Machine)** A TM \(U\) is said to be universal if for every input \((<M>,x)\) where \(<M>\) is the description of a Turing Machine \(M\), the output of \(U\) is the same as the output of \(M\) on input \(x\). (In particular, if \(M\) does not halt on \(U\) then neither does \(U\).)

To anyone familiar with computer programming, this is an obvious fact since one can write a C program to simulate any C program. (Such programs are called interpreters.)

1.3.1 The Church-Turing Thesis and its strong form.

The Church-Turing Thesis states that the Turing machine can simulate any physically implementable computational model. This is a hypothesis about the world rather than a claimed truth. It seems to us that every computational device must use some physical process to store information and manipulate it in discrete steps. The Turing machine (or a C program) should be able to simulate this by simulating the underlying physical process.

The CT thesis is reasonably well-accepted. However, it also has a stronger form, which is slightly more controversial: all physically implementable computational models can be simulated by a Turing machine with only polynomial slowdown. Namely, \(t\) steps on the model can be simulated in \(O(t^c)\) steps on the Turing machine, where \(c\) is a fixed constant depending upon the two models. This is a much stronger assumption about the physical world, but empirically it also seems to be true of all computational models we can conceive of. The one possible exception is a computational model based upon quantum mechanics that—as we will see in Chapter 21—may not be polynhomially equivalent to the Turing machine. However, we do not yet know if the quantum model is “physically implementable.” Furthermore, even if it is physically implementable, it is unclear whether it allows superpolynomial speedups for more than a small class of number-theoretic computations.
Though none of our results in this book rely on the strong form of the CT thesis, many of the definitions are motivated by it. Complexity classes such as P and NP have a strong intuitive appeal precisely because the thesis seems to be true (or largely true). Hence these classes have a universal meaning: if aliens in outer space would independently discover computation —using a model very different from the Turing machine— the classes P and NP would be the same for them as for us.

Chapter notes

The Turing Machine is the analog of any modern programming language — albeit one with no inbuilt prohibition about memory size. Though the assumption of an infinite memory may seem unrealistic at first, in the complexity setting it is of no consequence since we will restrict the machine to use a finite amount of tape cells (the number allowed will depend upon the input size).

The Turing Machine should be thought of as a logical construct, rather than as a piece of hardware. Most computers today are implementations of a universal computer using silicon chips. But many other physical phenomena can be used to construct universal TMs: amusing examples include bouncing billiards balls, cellular automata, and Conway’s Game of life.

Finally, it is also possible to study complexity theory axiomatically in a machine-independent fashion. See Cobham [Cob64] and Blum [Blu67] for two approaches.
Part I

Basic Complexity Classes
Chapter 2

P, NP and all that

“(if \( \phi(n) \approx Kn^2 \)) then this would have consequences of the greatest magnitude. That is to say, it would clearly indicate that, despite the unsolvability of the (Hilbert) Entscheidungsproblem, the mental effort of the mathematician in the case of the yes-or-no questions would be completely replaced by machines.... (this) seems to me, however, within the realm of possibility.”
Kurt Gödel in letter to John von Neumann, 1956; see [Sip92]

“In modern terminology, if SAT has a quadratic time algorithm

“I conjecture that there is no good algorithm for the traveling salesman problem. My reasons are the same as for any mathematical conjecture: (1) It is a legitimate mathematical possibility, and (2) I do not know.”
Jack Edmonds, 1966

“In this paper we give theorems that suggest, but do not imply, that these problems, as well as many others, will remain intractable perpetually.”
Richard Karp [Kar72], 1972

We start by defining the classes \( P \) and \( NP \). The question whether they are the same (the \( P \) versus \( NP \) problem) is the most important question in complexity theory and the motivation for much of what we will study.
We will usually use asymptotic notation in measuring the resources used by a TM. Let $\text{DTIME}(t(n))$ consist of every language that can be decided by a deterministic multitape TM whose running time is $O(t(n))$ on inputs of size $n$. Let $\text{NTIME}(t(n))$ be the class of languages that can be decided by a nondeterministic multitape TM (NDTM for short) in time $O(t(n))$.

**Definition 2.1**

$P = \bigcup_{c \geq 0} \text{DTIME}(n^c)$.  
$NP = \bigcup_{c \geq 0} \text{NTIME}(n^c)$

NP contains decision problems for which a “YES” answer has a short “certificate” whose size is polynomial in the input length, and which can be verified deterministically in polynomial time. Formally, we have the following definition, which is easily seen to be equivalent to Definition 2.1: one side follows by noting that an NDTM can “guess” a certificate in polynomial time and then verify it; the other side follows from thinking of the nondeterministic choices made by an NDTM as a certificate whose length is polynomial in the input size.

**Definition 2.2 (Alternative definition of NP)** Language $L$ is in NP if there is a language $L_0 \in P$ and constants $c, d > 0$ such that

$$\forall x \in \{0, 1\}^* \quad x \in L \iff \exists y \in \{0, 1\}^*, |y| \leq |x|^c + d \quad \text{and} \quad (x, y) \in L_0.$$  

**Example 2.1** The following languages are in NP:

- CLIQUE = $\{<G, k> : \text{graph } G \text{ has a clique of size } \geq k\}$, where a clique is a subset of vertices in which there is an edge between every pair. (The “certificate” for membership in this language is the clique.)
- 3-COLOR = $\{G : \text{graph } G \text{ has a coloring with 3 colors}\}$, where a coloring is an assignment of Red, Blue, Green (or 1, 2, 3) to each vertex such that no adjacent vertices get the same color.
- TRAVELING SALESMAN (decision version): Given a set of $n$ nodes, all $\binom{n}{2}$ internode distances $d_{ij}$, and a number $K$, decide if there is a closed circuit (“salesman tour”) that visits every node exactly once and has total length at most $K$. Here the certificate for a YES answer is any tour of length at most $K$.

These problems are not known to be in P.

The language $\{G : G \text{ is a connected graph}\}$ is in P, since a simple breadth first search can detect the connectivity of the graph.
2.0.2 The philosophical importance of $P$

The class $P$ is felt to capture the notion of decision problems with “feasible” decision procedures. Of course, one may argue whether $\text{DTIME}(n^{100})$ represents “feasible” computation in the real world. However, in practice, whenever we show that a problem is in $P$, we usually find an $n^3$ or $n^5$ time algorithm, not an $n^{100}$ algorithm. (It has also happened a few times that the first polynomial-time algorithm for a problem had high complexity, say $n^{20}$, but soon somebody simplified it to say an $n^5$ algorithm.)

Furthermore, we believe that the class $P$ is invariant to the choice of a computational model, since all “reasonable” computational models we can think of happen to be polynomially equivalent\footnote{This was earlier formalized as the Strong Form of the Church Turing Thesis. Recent results suggest that a computational model based upon quantum mechanics may not be polynomially equivalent to the Turing machine, though we do not yet know if this model is “reasonable” (i.e., can be built). We will discuss the quantum model later in the book.}. Namely, $t$ steps on one model can be simulated in $O(tc)$ steps on the other, where $c$ is a fixed constant depending upon the two models. Hence $P$ (as well as $NP$) seems to have a universal meaning: if aliens in outer space would independently discover computation —using a model very different from the Turing machine—the classes $P$ and $NP$ would be the same for them as for us.

Now we mention why $P$ is a natural class from the viewpoint of a programmer. Suppose an undergraduate programmer is asked to invent the definition of an “efficient” computation. Presumably, he/she would agree that a computation that runs in linear or quadractic time is “efficient.” Next, it is natural to consider the closure of this class of computations under composition. (Informally speaking, this means that if an program itself does “efficient” computations and it calls subroutines that are “efficient” then we consider it “efficient.”) This closure is exactly the class of polynomial-time computations (Cobham [Cob64]). Of course, Cobham’s result makes intuitive sense since composing a polynomial function with another polynomial function gives a function with polynomial growth (for every constants $c, d > 0$, $(n^c)^d = n^{cd}$) but the exact proof requires some care.

2.0.3 The philosophical importance of $NP$

At a totally abstract level, the $P$ versus $NP$ question may be viewed as a question about the power of nondeterminism in the Turing machine model. (Similar questions have been completely answered for simpler models such as finite automata.)
However, the “certificate” definition of \( \text{NP} \) also suggests that the \( \text{P} \) versus \( \text{NP} \) question captures a widespread phenomenon of some philosophical importance (and a source of great frustration to my undergraduate students): recognizing the correctness of an answer is often easier than coming up with the answer. To give other analogies from life: appreciating a Beethoven sonata is far easier than composing the sonata; verifying the solidity of a design for a suspension bridge is easier (to a civil engineer anyway!) than coming up with a good design; verifying the proof of a theorem is easier than coming up with a proof itself (a fact referred to in Gödel’s letter mentioned at the start of the chapter), and so forth. In such cases, coming up with the right answer seems to involve exhaustive search over an exponentially large set. The \( \text{P} \) versus \( \text{NP} \) question asks whether exhaustive search can be avoided in general. It seems obvious to most people—and the basis of many false proofs proposed by amateurs—that exhaustive search cannot be avoided: checking that a given salesman tour (provided by somebody else) has length at most \( K \) ought to be a lot easier than coming up with such a tour by oneself. (The careful reader may object that our discussion is conflating the decision problem of deciding whether or not a tour exists—an \( \text{NP} \) problem—with the corresponding search problem of finding the tour—which is not a decision problem and hence not in \( \text{NP} \); this issue is clarified in Section 2.4.) Unfortunately, turning this intuition into a proof has proved difficult.

Remark 2.1 Just as \( \text{NP} \) was defined to capture some phenomenon of practical or philosophical interest, the same is true for many other complexity classes encountered later in the book.

### 2.1 Reducibility and NP-completeness

A polynomial-time reduction from language \( A \) to language \( B \) is a polynomial-time computable function \( f \) mapping strings to strings, such that \( x \in A \) if and only if \( f(x) \in B \). If such a reduction exists we say \( A \) is polynomial-time reducible to \( B \), denoted \( A \leq_p B \). If \( A \leq_p B \) then the decision problem for \( A \) is “no harder” than the decision problem for \( B \). (See part 1 of Theorem 2.1 below.) Obviously, this holds only when we are interested in polynomial-time solvability: when studying other complexity measures we may require other notions of reducibility.

It is easily checked that polynomial-time reductions compose, namely, if \( A \leq_p B \) and \( B \leq_p C \), then \( A \leq_p C \). (This transitivity property is usually
2.1. REDUCIBILITY AND NP-COMPLETENESS

the minimum one expects from any reasonable notion of reducibility. We will later see many other notions of reducibility.)

A language is \textit{NP-hard} if there is a polynomial-time reduction from every NP language to it. An NP-hard language is \textit{NP-complete} if it is NP-hard and in NP. The following statements follow from the definition.

\textbf{Theorem 2.1}

1. If \( A \leq_p B \) and \( B \in P \) then \( A \in P \).

2. If language \( A \) is NP-complete, then \( A \in P \) iff \( P = NP \).

Thus to study the P versus NP question it suffices to study whether any NP-complete problem can be decided in polynomial time. All NP-complete problems are interreducible by definition, so we can pick our favorite one to work with. In this sense an NP-complete problem captures the essence of NP. (In many places in the book we will encounter other complexity classes and will try to define a notion of complete problems for them in an effort to capture the essence of those classes.)

Do NP-hard problems exist? There are some trivial examples, for instance, the well-known \textit{halting problem}, which is undecidable (i.e., there is no Turing machine that halts with the correct answer on every input)

\[
A_{TM} = \{ < M, w > : \text{Turing machine } M \text{ halts when run on input } w \} \tag{2.1}
\]

is NP-hard. (See Exercises.) Similarly, there are trivial examples of NP-complete languages, such as the following one:

\[
K_{TM,poly} = \{ < M, w, 1^n > : \text{NDTM } M \text{ accepts } w \text{ in time } n \} . \tag{2.2}
\]

However, such NP-complete problems are not very useful because their definition is intimately tied to the notion of the Turing machine, and hence the fact that they are NP-complete does not provide much new insight.

Around 1971, Cook [Coo71] and Levin [Lev73] in independent papers gave examples of combinatorial NP-complete problems whose definition seems to have nothing to do with Turing machines. (These papers actually also introduced the notion of NP-completeness.) A subsequent paper of Karp showed that NP-completeness occurs widely: many combinatorial problems of practical interest are NP-complete. These papers initiated the study of NP-completeness. To date, thousands of computational problems in a variety of disciplines have been found to be NP-complete.
Some of the simplest examples of NP-complete problems come from propositional logic. A 3CNF formula in propositional logic is a formula in some variables boolean variables $x_1, x_2, x_3, \ldots$ of the form

$$\bigwedge_i (y_{i1} \lor y_{i2} \lor y_{i3}),$$

where each $y_{ij}$ is either a variable or its negation. (Following usual convention, we equate the boolean values True and False with the bits 1 and 0 respectively.)

3SAT is the set of boolean 3CNF formulae that have a satisfying assignment (that is, an assignment to the variables that makes it evaluate to true).

### 2.1.1 The Cook-Levin Theorem: Computation is Local

Both Cook and Levin showed that 3SAT is NP-complete. We briefly recall this classical reduction. Let $L$ be an NP language that is recognized by an NDTM $M$ in $n^c$ time. Let $x$ be an input and $n = |x|$. The reduction uses the idea of a tableau, which is a step-by-step transcript whose $i$th line contains the configuration of the tape at step $i$ of the computation. Let $Q$ be the (finite) set of states of $M$ and $\Gamma$ the tape alphabet. For ease of notation assume the machine has a single tape. Then the state at any step is represented by a string $\alpha q \beta$ where $q \in Q$ is the state that $M$ is in, $\alpha, \beta \in Z^*$ are the strings on the tape to the left and right respectively of the machine’s head.

Clearly, $x \in L$ iff there exists a tableau of the computation of $M$ that contains $x$ in the first line and in which the last line shows that $M$ accepts (Figure 2.1).

![Figure 2.1: Tableau as used in Cook-Levin reduction.](image-url)
Since the computation runs for $n^c$ steps, it uses at most $n^c$ cells on the tape, so the tableau is an $n^c \times n^c$ matrix.

The reduction produces a boolean formula $\varphi_x$ that has, for each cell $(i, j)$ of the tableau, $|Q| + |\Gamma|$ variables denoted by $X_{i,j,\sigma}$ where $\sigma \in Q \cup \Gamma$. The satisfying assignments to $\varphi_x$ will turn out to be in 1-to-1 correspondence with tableaus of accepting computations of $M$ on input $x$.

We will write $\varphi_x$ as

$$\phi_{\text{syntax}} \land \phi_{\text{start}} \land \phi_{\text{accept}} \land \phi_{\text{compute}}.$$ 

The first subformula $\phi_{\text{syntax}}$ says that for each cell in the tableau, exactly one variables gets a truth value 1. Thus

$$\phi_{\text{syntax}} = \bigwedge_{1 \leq i,j \leq n^c} \left( \bigvee_{\sigma} X_{ij,\sigma} \land \bigwedge_{\sigma \neq \sigma'} \left( X_{ij,\sigma} \lor X_{ij,\sigma'} \right) \right).$$

Subformula $\phi_{\text{start}}$ says that the tableau’s first line contains $x$ and $\phi_{\text{accept}}$ says that $M$ is in an accept state in the last line; these are left to the reader. The main insight is in writing $\phi_{\text{compute}}$, which says that the transcript depicted in the tableau is a computation performed according to the program of $M$. Expressing this succinctly relies on the fact that a Turing machine computation is very local. When going from one step to the next, nothing much changes on the tape except contents of the cell underneath the head. Thus the tableau represents a correct computation iff all $2 \times 3$ windows look “correct,” i.e. they satisfy some local consistency conditions: if the head was on a cell inside the window, then the contents of that cell should have been modified according to $M$’s finite description, and if not then the cell contents should be unchanged. There are only a finite number of such consistency conditions for each window and it is easy to represent these with a finite boolean formula in the variables corresponding to that window. Thus one can write a formula of the type

$$\bigwedge_{1 \leq i,j \leq n^c} \Psi_{ij},$$

where $\Psi_{ij}$ is a formula in $X_{rs,\sigma}$ for $r = i - 1, i, i + 1$ and $s = j, j + 2$ and $\sigma \in Q \cup \Gamma$ (in other words, the variables belonging to the window at $(i, j)$) expressing that the values to these variables correspond to a “correct” window. Since the number of windows is $n^{2c}$, this formula has size $O(n^{2c})$. 
Remark 2.2 The Cook-Levin theorem is a good example of the power of abstraction. Even though the theorem holds regardless of whether our computational model is the C programming language or the Turing machine, it may have been considerably more difficult to discover in the former context.

2.1.2 The web of reductions

Cook and Levin had to show how every NP language can be reduced to 3SAT. To prove the NP-completeness of any other language, we do not need to work as hard: it suffices to reduce 3SAT to this language. This follows from the following easy observation.

Theorem 2.2
If languages $A$ is NP-hard and $A \leq_p B$ then $B$ is also NP-hard.

Thus if $B \in$ NP and 3SAT $\leq_p B$ then $B$ is NP-complete. Thus reductions from 3SAT are used to prove new NP-completeness results, which in turn are used to prove further NP-completeness results. This web of reductions and thousands of NP-complete problems are described in Garey and Johnson’s classic 1979 text.

2.2 coNP

The class coNP is also very relevant to the study of the P versus NP question.

Definition 2.3 $\text{coNP} = \{L : \overline{L} \in P\}$.

The following is an example of a coNP language:

$\overline{\text{SAT}} = \{<\varphi> : \varphi \text{ is not satisfiable}\}$.

Undergraduates sometimes mistakenly convince themselves that $\text{SAT} \in \text{NP}$. They have the following polynomial time NDTM in mind. On input $\varphi$, the machine guesses an assignment. If this assignment satisfies $\varphi$, the machine outputs REJECT and otherwise it outputs ACCEPT. This NDTM does not do the job: indeed it accepts every unsatisfiable $\varphi$ but in addition it also accepts many satisfiable formulae.

That is why pedagogically, I prefer the following definition of coNP, which is easily checked to be equivalent to first.
2.3 EXPTIME AND NEXPTIME

Definition 2.4 (Alternative definition of coNP) Language \( L \) is in coNP if there is a language \( L_0 \in P \) and constants \( c,d > 0 \) such that

\[
\forall x \in \{0,1\}^* \quad x \in L \iff \forall y \in \{0,1\}^*, |y| \leq |x|^c + d \quad \text{and} \quad (x,y) \in L_0.
\]

The key fact to note is the use of "\( \forall \)" in this definition where Definition 2.2 used \( \exists \).

We can define coNP-completeness in analogy to NP-completeness: a language is coNP-complete if it is in coNP and every coNP language is polynomial-time reducible to it.

Example 2.2 The following language is coNP-complete:

TAUTOLOGY = \{ \varphi : \varphi \text{ is a boolean formula that is satisfied by every assignment} \}.

It is in coNP (the language \( L_0 \) is the set of pairs \( (< \varphi >, s) \) where \( \varphi \) is a boolean formula and \( s \) is a string that is a satisfying assignment.

For every \( L \in \text{coNP} \) we have to show that \( L \leq_p \text{TAUTOLOGY} \). But this is easy: just modify the Cook-Levin reduction from \( \overline{L} \) (which is in \( \text{NP} \)) to \( \text{SAT} \). For every input \( x \in \{0,1\}^* \) that reduction produces a formula \( \phi_x \) that is satisfiable iff \( x \in \overline{L} \). Now consider the formula \( \neg \phi_x \). It is in TAUTOLOGY iff \( x \in L \), and this completes the description of the reduction.

It is a simple exercise to check that if \( P = \text{NP} \) then \( \text{NP} = \text{coNP} = P \). Put contrapositively, if we can show that \( \text{NP} \neq \text{coNP} \) then we have shown \( P \neq \text{NP} \). Most researchers believe that \( \text{NP} \neq \text{coNP} \). The intuition is almost as strong as for the \( P \) versus \( \text{NP} \) question: it seems hard to believe that there is any short certificate for certifying that a given formula is a TAUTOLOGY, in other words, to certify that every assignment satisfies the formula.

2.3 EXPTIME and NEXPTIME

The following two classes are exponential time analogues of \( P \) and \( \text{NP} \).

Definition 2.5 \( \text{EXPTIME} = \bigcup_{c \geq 0} \text{DTIME}(2^{n^c}) \).
\( \text{NEXPTIME} = \bigcup_{c \geq 0} \text{NTIME}(2^{n^c}) \).
Note that $\text{P} \subseteq \text{NP} \subseteq \text{EXPTIME} \subseteq \text{NEXPTIME}$. Is there any point to studying classes involving exponential running times? The following simple result —providing merely a glimpse of the rich web of relations we will be establishing between disparate complexity questions— may be a partial answer. It also introduces a technique called padding, whereby we transform a language by “padding” every string in a language with a string of (useless) symbols. This simple idea underlies many elementary results in complexity theory.

**Theorem 2.3**

If $\text{EXPTIME} \neq \text{NEXPTIME}$ then $\text{P} \neq \text{NP}$.

**Proof:** We prove the contrapositive: assuming $\text{P} = \text{NP}$ we show $\text{EXPTIME} = \text{NEXPTIME}$. Suppose $L \in \text{NTIME}(2^{cn})$ and NDTM $M$ decides it. We claim that then the language

$$L_{\text{pad}} = \left\{ <x, 1^{2^{|x|c}}>: \ x \in L \right\} \quad (2.3)$$

is in $\text{NP}$. Here is an NDTM for $L_{\text{pad}}$: given $y$, first check if there is a string $z$ such that $y = <z, 1^{2^{|z|c}}>$. If not, output REJECT. If $y$ is of this form, then run $M$ on $z$ for $2^{|z|c}$ steps and output its answer. Clearly, the running time is polynomial in $|y|$, and hence $L_{\text{pad}} \in \text{NP}$. Hence if $\text{P} = \text{NP}$ then $L_{\text{pad}}$ is in $\text{P}$. But if $L_{\text{pad}}$ is in $\text{P}$ then $L$ is in $\text{EXPTIME}$: to determine whether an input $x$ is in $L$, we just pad the input and decide whether it is in $L_{\text{pad}}$ using the polynomial-time machine for $L_{\text{pad}}$. □

We note that most of the complexity classes studied later are, like $\text{P}$ and $\text{NP}$, also contained in $\text{EXPTIME}$ or $\text{NEXPTIME}$.

### 2.4 Decision versus search

We have chosen to define the notion of $\text{NP}$ using Yes/No problems (“Is the given formula satisfiable?”) as opposed to search problems (“Find a satisfying assignment to this formula if one exists”). However, it is easy to see that if we can solve the decision problem in time $T(n)$ then we can solve the search problem in time $n^cT(n)$ for some small constant $c$. This fact is true for all $\text{NP}$-complete languages but let us illustrate it using 3SAT.

Suppose $A$ is an algorithm that decides 3SAT in time $T(n)$. Given a 3CNF formula $\varphi$ with $n$ variables, we can use $A$ to first check that $\varphi$ is satisfiable. If so, we substitute $x_1 = 0$ and $x_1 = 1$ in $\varphi$ (this simplifies and shortens the formula a little and leaves a formula with $n - 1$ variables) and
check which of the two is satisfiable (it is possible that both are). Say it is the first. Then we fix $x_1 = 0$ from now on and continue with the simplified formula. Continuing this way we end up fixing all $n$ variables while ensuring that each intermediate formula is satisfiable. Thus the final assignment to the variables satisfies $\varphi$. Clearly, the running time is $n^2T(n)$ or so.

Formally, we say that 3SAT is downward self-reducible: the solution of instances with $n$ variables reduces to the solution of instances with $n - 1$ variables.

## 2.5 List of some problems in $P$ and $NP$

The following problems are in $P$. We mention these because it is somewhat nontrivial to show that they are in $P$.

### LINEAR EQUATIONS:

$$= \{ < A \in \mathbb{Q}^{m \times n}, b \in \mathbb{Q}^m > : \exists x \in \mathbb{Q}^n, Ax = b \}.$$  
Linear systems of equations over any field can be solved in polynomial time (using Gaussian elimination for example). The slightly tricky part in the analysis is to show that the numbers produced in the intermediate calculations have size polynomial in the input size. (Put this in; or show it is in $NP$)

### LP:

$$= \{ < A, b, c, C > : A \in \mathbb{Q}^{m \times n}, b \in \mathbb{Q}^m, c \in \mathbb{Q}^n, C \in \mathbb{Q} \text{ such that } \exists x \geq 0, Ax \leq b, c^T x \geq C \}.$$  
(Note that if $y, z$ are vectors then $z \geq y$ if each coordinate of $z$ is greater than the corresponding coordinate of $y$.) It was a longstanding open problem whether this is in $P$, until Khachiyan’s invention of the Ellipsoid algorithm for Linear Programming in 1979.

### PRIMES:

$$= \{ p \in \mathbb{Z} : p \text{ is a prime integer} \}.$$  
Note that the trivial algorithm — check all potential factors up to $\sqrt{p}$ — takes around $\sqrt{p}$ time, which is exponential in the size of input, $\lceil \log p \rceil$. Furthermore, the complement of the language, the set of composite integers, is in $NP$ since a factor can be guessed (if one exists) in $O(\log p)$ time. Hence PRIMES $\in \text{coNP}$. Pratt showed in 1974 that actually PRIMES $\in NP \cap \text{coNP}$. Soon after, randomized polynomial time algorithms were discovered for this problem (we will study randomized algorithms in Chapter 7). In 2002, Agarwal et al. discovered a deterministic algorithm, thus showing PRIMES $\in P$.

### NP optimization problems
CHAPTER 2. \textit{P}, \textit{NP} AND ALL THAT

2.6 More thoughts about \textit{P}, \textit{NP}, and all that

2.6.1 Why 3SAT?

The reader may wonder why complexity theorists are so fond of 3SAT. One answer is that it is useful for proving the \textit{NP}-completeness of other problems: it has very minimal combinatorial structure and thus easy to use in reductions. Another answer has to do with history: propositional logic has had a central role in mathematical logic — in fact it was exclusively the language of classical logic (e.g., in ancient Greece). This historical resonance is one reason why Cook and Levin were interested in 3SAT. A third answer has to do with practical importance: it is a simple example of \textit{constraint satisfaction problems}, which are ubiquitous in many fields including artificial intelligence.

2.6.2 What if \textit{P} = \textit{NP}?

If \textit{P} = \textit{NP} — specifically, if an \textit{NP}-complete problem like 3SAT had a very efficient algorithm running in say $O(n^2)$ time — then the world would be mostly a Utopia. Mathematicians could be replaced by efficient theorem-discovering programs (a fact pointed out in Kurt Gödel’s 1956 letter and discovered three decades later). This follows from the fact that the following problem is in \textit{NP}:

$$\text{THEOREMS} = \{(\varphi, 1^n) : \varphi \text{ has a formal proof of length } \leq n \text{ in axiomatic set theory}\}.$$  

(The “axiomatic set theory” could be replaced by any other mathematical or scientific area with an axiomatic description. The exact choice of axioms for doesn’t matter so long as they have a succinct description, which currently accepted axiomatic systems do.)

In general for every search problem whose answer can be efficiently verified (or has a short certificate of correctness), we will be able to find the correct answer or the short certificate in polynomial time. AI software would be perfect since we could easily do exhaustive searches in a large tree of possibilities. Inventors and engineers would be greatly aided by software packages that can design the perfect part or gizmo for the job at hand. VLSI designers will be able to whip up optimum circuits, with minimum power...$

\footnote{In fact, the exercises ask you to prove that the problem is \textit{NP}-complete — something Gödel could not have realized — and hence the \textit{P} versus \textit{NP} question is a \textit{rephrasing} of Gödel’s question, which asks whether or not there is a algorithm that finds mathematical proofs in time polynomial in the length of the proof.}
requirements. (This will be a consequence of our study of the Polynomial Hierarchy in Chapter 5.)

Somewhat intriguingly, this Utopia would have no need for randomness. As we will later see, if \( P = NP \) then randomized algorithms would buy essentially no efficiency gains over deterministic algorithms; see Chapter 7. (Armchair philosophers should ponder this.)

This Utopia would also come at one price: there would be no privacy in the digital domain. Any encryption scheme would have a trivial decoding algorithm. There would be no digital cash, no PGP, no RSA (as we will later see). We would just have to learn to get along better without these, folks.

### 2.6.3 In praise of reductions

Though originally invented as part of the theory of \( NP \)-completeness, the polynomial-time reduction (together with its first cousin, the randomized polynomial-time reduction defined in Section 7.6) has led to a rich understanding of complexity above and beyond \( NP \)-completeness. Much of complexity theory and cryptography today (thus, many chapters of this book) consists of using reductions to make connections between disparate complexity theoretic conjectures.

A probable explanation is that a reduction is nothing but an algorithm for transforming one problem into another, and humans just seem better at designing algorithms than at proving lowerbounds on Turing machines.

### Exercises

§1 Show that the languages defined in (2.1) and (3.2) are \( NP \)-hard and \( NP \)-complete respectively.

§2 If \( P = NP \) then there is a polynomial time decision algorithm for \( 3\text{SAT} \). Show that in fact if \( P = NP \) then there is also a polynomial time algorithm that, given any \( 3\text{CNF} \) formula, produces a satisfying assignment if one exists.

§3 We have defined a relation \( \leq_p \) among languages. We noted that it is reflexive (that is, \( A \leq_p A \) for all languages \( A \)) and transitive (that is, if \( A \leq_p B \) and \( B \leq_p C \) then \( A \leq_p C \)). Show that it is not commutative, namely, \( A \leq_p B \) need not imply \( B \leq_p A \).

§4 Suppose \( L_1, L_2 \in NP \). Then is \( L_1 \cup L_2 \) in \( NP \)? What about \( L_1 \cap L_2 \)?
§5 Mathematics can be axiomatized using for example the Zermelo Frankel system, which has a finite description. Argue at a high level that the following language is NP-complete.

\{ < \varphi, 1^n > : \text{math statement } \varphi \text{ has a proof of size at most } n \text{ in the ZF system} \}.

(Hints: Why is this language in NP? Is boolean satisfiability a mathematical statement?) Conclude that if P = NP then mathematicians can be replaced by polynomial-time Turing machines.

§6 Show that \( \text{NP} = \text{coNP} \) iff 3SAT and TAUTOLOGY are polynomial-time reducible to one another.

§7 Can you give a definition of NEXPTIME analogous to the definition of NP in Definition 2.2? Why or why not?

§8 We say that a language is NEXPTIME-complete if it is in NEXPTIME and every language in NEXPTIME is polynomial-time reducible to it. Describe a NEXPTIME-complete language.

§9 Show that if \( L \in \text{NTIME}(t(n)) \) then we can give a reduction from \( L \) to 3SAT that transforms instances of size \( n \) into 3CNF formulae of size \( O(t(n) \log t(n)) \). (Hint: The representation of the tableau used in our description of the Cook-Levin reduction is very inefficient, since all the action happens only near the machine’s head.)

§10 The notion of polynomial-time reducibility used in Cook’s paper was somewhat different: a language \( A \) is Cook-reducible to language \( B \) if there is a polynomial time TM \( M \) that, given an oracle for deciding \( B \), can decide \( A \). (An oracle for \( B \) is a magical extra tape given to \( M \), such that whenever \( M \) writes a string on this tape and goes into a special “invocation” state, then the string —in a single step!—gets overwritten by 1 or 0 depending upon whether the string is or is not in \( B \).)

Show that 3SAT is Cook-reducible to TAUTOLOGY.

§11 (Berman’s theorem 1978) A language is called unary if every string in it is of the form \( 1^i \) (the string of \( i \) ones) for some \( i > 0 \). Show that if a unary language is NP-complete then \( P = NP \). (Hint: If there is a \( n^c \) time reduction from 3SAT to a unary language \( L \), then this reduction can only map size \( n \) instances of 3SAT to some string of the form \( 1^i \) where \( i \leq n^c \). Now consider the trivial —and exponential-time—
2.6. MORE THOUGHTS ABOUT P, NP, AND ALL THAT

backtracking algorithm for 3SAT and use the above fact to adaptively prune the tree of all possibilities.)

§12 In the VERTEX COVER problem we are given an undirected graph $G$ and an integer $K$ and have to decide whether there is a subset $S$ of at most $K$ vertices such that for every edge $\{i, j\}$ of $G$, at least one of $i$ or $j$ is in $S$. Prove that this problem is NP-complete.

§13 In the MAX CUT problem we are given an undirected graph $G$ and an integer $K$ and have to decide whether there is a subset of vertices $S$ such that there are at least $K$ edges that have one endpoint in $S$ and one endpoint in $\overline{S}$. Prove that this problem is NP-complete.

§14 Suppose $L_1, L_2 \in \text{NP} \cap \text{coNP}$. Then show that $L_1 \oplus L_2$ is in $\text{NP} \cap \text{coNP}$, where

$$L_1 \oplus L_2 = \{x : x \text{ is in exactly one of } L_1, L_2\}.$$ 

Chapter notes and history

We omitted a detailed discussion of formal complexity, and in particular the fact that the class $\text{DTIME}(f(n))$ can be paradoxical if $f$ is not a proper complexity function. (see the standard text [HMU01]). We say $f$ is proper if $f(n) \geq f(n - 1)$ and there is a TM that on input $x$ outputs a string of length $f(|x|)$ using time $O(|x| + f(|x|))$ and space $O(f(|x|))$. This notion will reappear in Chapter 4.

Sipser’s survey [Sip92] succinctly describes the evolution of the concepts of P, NP in various scholarly articles from the 1950s and 1960s. It also contains a translation of Gödel’s remarkable letter, which was discovered among von Neumann’s papers in the 1980s.

The “TSP book” by Lawler et al. [LLKS85] also has a similar chapter, and it traces interest in the Traveling Salesman Problem back to the 19th century. Furthermore, a recently discovered letter by Gauss to Schumacher shows that Gauss was thinking about methods to solve the famous Euclidean Steiner Tree problem — today known to be NP-hard — in the early 19th century.
Chapter 3

Space complexity

“(our) construction... also suggests that what makes “games” harder than “puzzles” (e.g. NP-complete problems) is the fact that the initiative (“the move”) can shift back and forth between the players.”
Shimon Even and Robert Tarjan, 1976

This chapter tries to study the memory requirements of computational tasks. We define space-bounded computation, which has to be performed by the TM using a restricted number of tape cells. A space-bounded machine has two tapes. The input is provided on a read-only tape, which, as the name suggests, can only be read and not overwritten. The machine also has a read-write work tape. We say that the machine runs in $S(n)$ space if it can perform its computations using $O(S(n))$ cells of the work tape whenever the input has $n$ bits. (See Figure 3.1.) We denote by SPACE($S(n)$) the class of languages that can be decided in $O(S(n))$ space. We can similarly define nondeterministic space-bounded machines, and the class NSPACE($s(n)$). We will restrict attention to $S(n) \geq \log n$, so the machine has enough space to remember the location of its head in the input tape. (Machines that use less than $\log n$ space have been studied but are usually not very interesting; see exercises.) Note that DTIME($t(n)$) $\subseteq$ SPACE($t(n)$) since a TM can access only one tape cell per step.

**Theorem 3.1**

$SPACE(s(n)) \subseteq NSPACE(s(n)) \subseteq DTIME(2^{O(s(n))}).$

**Proof:** We first see why $SPACE(s(n)) \subseteq DTIME(2^{O(s(n))}).$ A machine with a work tape of size $s(n)$ only has $O(n \cdot 2^{O(s(n))}) = 2^{O(s(n))}$ different
configurations (reason: once we fix the input on the read-only tape, the machine’s configuration is completely described by the position of its head on the input tape, the machine’s inner state, and the contents of the work tape) and the machine cannot enter the same configuration twice because that would mean it is in an infinite loop. (Recall that the machine is required to halt on every input.)

It is easy to extend the above insight to a proof that \( \text{NSPACE}(s(n)) \subseteq \text{DTIME}(2^{O(s(n))}) \). If \( L \in \text{NSPACE}(s(n)) \) is decided by an NDTM \( M \) then given input \( x \) of size \( n \) we can produce the graph whose nodes are all possible configurations of \( M \) on \( x \). As noted, this has size \( 2^{O(s(n))} \). The graph has an edge from configuration \( C_1 \) to configuration \( C_2 \) iff \( C_2 \) can be obtained from \( C_1 \) in one step. Thus \( M \) accepts \( x \) iff there is a path in this graph from the start configuration to some accepting configuration, which is easy to check using breadth-first search in \( 2^{O(s(n))} \) time.

The above proof serves to remind us that space \( S(n) \) machines may run for \( 2^{O(S(n))} \) time (a simple example is the machine that uses its work tape to maintain a counter which it increments from 1 to \( 2^{S(n)-1} \) and then halts). However, it is an open problem whether there is a language in \( \text{DTIME}(2^{O(S(n))}) \) for some \( S(n) > \log n \) that is not in \( \text{SPACE}(S(n)) \).

The following definitions are similar to the definitions of \( \text{P} \) and \( \text{NP} \).

**Definition 3.1**

\[ \text{PSPACE} = \cup_{c>0} \text{SPACE}(n^c). \]

\[ \text{NPSPACE} = \cup_{c>0} \text{NSPACE}(n^c). \]

Note that \( \text{NP} \subseteq \text{PSPACE} \), since polynomial space—in fact, linear space—is enough to decide \( 3\text{SAT} \) (just use the linear space to cycle through all \( 2^n \) assignments in order, where \( n \) is the number of variables, and note that once we have checked an assignment we can erase it from the work tape and reuse the work tape to check the next assignment).

Space-bounded computation continues to be interesting for \( S(n) < n \), of course, and the following classes are the most interesting in this context.

**Definition 3.2**

\[ \text{L} = \text{SPACE}(\log n). \]

\[ \text{NL} = \text{NSPACE}(\log n). \]

**Example 3.1** The reader should check (using the gradeschool method for arithmetic) that the following languages are in \( \text{L} \):

\[ \text{EVEN} = \{ x : x \text{ has an even number of 1s} \}. \]

\[ \text{MULT} = \{ (< n >, < m >, < nm >) : n \in \mathbb{N} \}. \]
We seem unable to do very complicated computations $O(\log n)$ space. Even so, we cannot currently rule out that $3\text{SAT} \in L$ (in other words —see the exercises— it is open whether $\text{NP} \neq \text{L}$).

Space-bounded computations with space $S(n) \ll n$ seem relevant to computational problems such as web crawling. The world-wide-web may be viewed crudely as a directed graph, whose nodes are web pages and edges are hyperlinks. Web crawlers seek to explore this graph for all kinds of information. The following problem is natural in this context.

**Example 3.2** Consider the following language.

$$\text{PATH} = \{ <G, s, t> : G \text{ is a directed graph in which there is a path from } s \text{ to } t \}$$

(3.1)

We claim that $\text{PATH} \in \text{NL}$. The reason is that a nondeterministic machine can take a “nondeterministic walk” starting at $s$, always maintaining the index of the vertex it is at, and using nondeterminism to select a neighbor of this vertex to go to next. The machine accepts iff the walk ends at $t$ in at most $n$ steps, where $n$ is the number of nodes. If the nondeterministic walk has run for $n$ steps already and $t$ has not been encountered, the machine rejects. The work tape only needs to hold $O(\log n)$ bits of information at any step, namely, the number of steps that the walk has run for, and the identity of the current vertex.

Is $\text{PATH} \in \text{L}$ as well? This is an open problem, which, as we will shortly see, is equivalent to whether or not $\text{L} = \text{NL}$. The reason is that $\text{PATH}$ captures the “essence” of NL (just as $3\text{SAT}$ captures the “essence” of NP).

Recent results show that the $\text{PATH}$ problem restricted to undirected graphs can be solved in $\log^{4/3} n$ space. We will talk about this more in Chapter 7.
CHAPTER 3. SPACE COMPLEXITY

3.1 PSPACE completeness

As already indicated, we do not know if \( P \neq \text{PSPACE} \), though we strongly believe that the answer is YES. Notice, \( P = \text{PSPACE} \) implies \( P = \text{NP} \).

Since complete problems can help capture the essence of a complexity class, we try to define complete problems for \( \text{PSPACE} \).

**Definition 3.3** A language \( A \) is \( \text{PSPACE} \)-hard if for every \( L \in \text{PSPACE} \), \( L \leq_p A \). If in addition \( A \in \text{PSPACE} \) then \( A \) is \( \text{PSPACE} \)-complete.

Clearly, a \( \text{PSPACE} \)-complete language is in \( P \) iff \( P = \text{PSPACE} \), so such a language may be viewed as the “most difficult” problem of \( \text{PSPACE} \).

As in the case of \( \text{NP} \), it is trivial to describe a \( \text{PSPACE} \)-complete problem. The following is one (Exercise):

\[
K_{TM,\text{pspace}} = \{ <M, w, 1^n>: \text{NDTM } M \text{ accepts } w \text{ in space } n \}. 
\]  

(3.2)

Now we describe a more nontrivial complete problem. A quantified boolean formula has the form \( Q_1 x Q_2 x_2 \cdots Q_n x_n \varphi(x_1, x_2, \ldots, x_n) \) where each \( Q_i \) is one of the two quantifiers \( \forall \) or \( \exists \) and \( \varphi \) is a (unquantified) boolean formula\(^1\).

Let \( \text{TQBF} \) be the set of quantified boolean formulae that are true.

**Example 3.3** The formula \( \forall x \exists y (x \land y) \lor (\overline{x} \land \overline{y}) \) (this is the same as \( \forall x \exists y (x \neq y) \)) is in \( \text{TQBF} \) but \( \forall x \forall y (x \land y) \lor (\overline{x} \land \overline{y}) \) is not in \( \text{TQBF} \). Note that here \( \forall \) and \( \exists \) quantify over the set \( \{0, 1\} \).

**Theorem 3.2**

\( \text{TQBF} \) is \( \text{PSPACE} \)-complete.

---

\(^1\)We are restricting attention to quantified boolean formulae which are in prenex normal form, i.e., all quantifiers appear to the left. From a complexity viewpoint this is without loss of generality since deciding the truth of general quantified boolean formulae is polynomial-time reducible to deciding the truth of this special kind.
3.1. PSPACE COMPLETENESS

PROOF: First we show TQBF \( \in \) PSPACE. If the input is

\[ \exists x_1 \forall x_2 \exists x_3 \ldots Q_n x_n \varphi(x_1, x_2, \ldots, x_n) \]

then we can think of all boolean assignments to the variables as constituting a complete binary tree of depth \( n \) (see figure). We can label the tree leaves by a truth value, namely, whether or not \( \varphi \) is satisfied by the assignment described by the path leading to that leaf. This tree may remind the reader of SAT, except instead of asking whether or not there is a leaf with label 1, here we are interested in inductively assigning a truth value to each node as follows. The leaves already have a boolean value. If an internal node corresponds to a variable quantified with \( \exists \), then it receives a value 1 iff at least one of its two children receives a value 1. If the node corresponds to a variable quantified with \( \forall \), then it receives a value 1 iff both of its two children receives a value 1. The formula is true iff the root node gets a value 1. This tree evaluation can be done by a simple recursive procedure, which when evaluating an internal node only needs to store the sequence of values defined on the path from the root to this node, i.e., at most \( n \) bits. Thus we have shown TQBF \( \in \) PSPACE.

Now for any \( L \in \) PSPACE we show that \( L \leq_p \) TQBF. Let \( M \) be a machine that decides \( L \) in \( n^c \) space. We show how to construct a quantified boolean formula of size roughly \( O(n^{2c}) \) that is true iff \( M \) accepts \( x \). The construction is reminiscent of the proof of the Cook-Levin theorem. The complication is that \( M \)'s computation could take \( 2^{O(n^c)} \) time, so there are too few variables to represent a tableau directly.

For notational convenience, we will group variables into vectors. A variable vector \( C \) has \( O(n^c) \) variables such that assigning boolean values to them gives a potential configuration of \( M \). (Many assignments—in fact, most—may yield nonsensical configurations.) An assignment vector \( D \) is an assignment of boolean values to a variable vector.

Restatement of Cook-Levin construction: For any variable vectors \( C_1, C_2 \) we can write in \( O(n^{c+1}) \) time an unquantified boolean formula \( \varphi_{CL}(C_1, C_2) \) such that for any assignment vectors \( D_1, D_2, \varphi_{CL}(D_1, D_2) = 1 \) iff \( D_1, D_2 \) correspond to valid configurations of \( M \) and \( D_2 \) can be obtained from \( D_1 \) in one step.

The main idea in the current proof is as follows. We show how to write for any \( T > 0 \) a quantified formula \( \psi_T(C_1, C_2) \) whose free—i.e. unquantified—variables are \( C_1, C_2 \) (the formula may have many other variables but they are all quantified) such that for any assignment vectors \( D_1, D_2, \psi_T(D_1, D_2) \)
is true iff $D_1, D_2$ correspond to valid configurations of $M$ and $D_2$ can be obtained by starting $M$ in $D_1$ and running it for $\lceil T \rceil$ steps.

Clearly, if $T = 0$ then $\psi_0(C_1, C_2)$ just asserts that $C_1, C_2$ are bit-wise the same vector. If $T = 1$, then $\psi_1(C_1, C_2) = \varphi_{CL}(C_1, C_2)$ fits the bill. For general $T$ we note that $C_2$ can be obtained from $C_1$ in $T$ steps iff there is an intermediate configuration $C_3$ such that $C_3$ can be obtained from $C_1$ in $\lceil T/2 \rceil$ steps and $C_2$ can be obtained from $C_3$ in $\lceil T/2 \rceil$ steps.

Figure 3.2: A partial tableau. Configuration $C_1$ leads to configuration $C_2$ in $T$ steps iff there is an intermediate configuration $C_3$ at the halfway point.

Thus it is tempting to write the formula

$$\psi_T(C_1, C_2) = \exists C_3 \psi_{T/2}(C_1, C_3) \land \psi_{T/2}(C_3, C_2).$$

(3.3)

This does the job logically speaking, but is too long: its length is $O(Tn^c)$ whereas we were hoping the length will be $O(n^c \log T)$. The correct expression for $\psi_T(C_1, C_2)$ involves reusing the same subformula to assert the fact that $C_1$ leads to $C_3$ in $\lceil T/2 \rceil$ steps and that $C_3$ leads to $C_1$ in $\lceil T/2 \rceil$ steps. It is as follows:

$$\exists C_3 \forall(C_4, C_5) \quad ((C_4, C_5) = (C_1, C_3) \lor (C_4, C_5) = (C_3, C_2)) \Rightarrow \psi_{T/2}(C_4, C_5).$$

(3.4)

The final reduction consists of the following. First, compute an upperbound $T = 2^{dn^c}$ for an appropriate constant $d$ (derived from the number of distinct configurations of $M$) on the running time of $M$ on input $x$. Then construct the quantified formula $\psi_{2dn^c}(C_1, C_2)$. Finally, substitute values for the free variables $C_1, C_2$ to obtain $\psi_{2dn^c}(D_{\text{start}}, D_{\text{accept}})$, where $D_{\text{start}}$ is the assignment vector corresponding to the start configuration (assume the machine is started with both heads at the left end of the tape) and $D_{\text{accept}}$ is the assignment vector corresponding to the accept configuration, which we assume is unique (just modify the machine’s description so that it erases fills
3.1. **PSPACE COMPLETENESS**

the work tape with 0’s and moves both heads to the tapes’ left ends before entering the accept state).

The resulting formula is not strictly a quantified boolean formula, on two counts. First, our definition of quantified boolean formulae requires all variables to be quantified before the boolean formula begins, whereas here (after unwrapping the recursive definition) we find quantifiers in the midst of the boolean formula. But it is an easy exercise to move these quantifiers to the beginning. Second, for ease of exposition we used the nonstandard boolean connectives $\Rightarrow$ and $=$ in the formula, but they can be easily expressed using the standard boolean connectives $\lor$, $\land$, $\neg$ by noticing that $p \Rightarrow q$ is equivalent to $\neg p \lor q$ and $p = q$ is equivalent to $(p \land q) \lor (\neg p \land \neg q)$.

The reduction applies these transformations on $\psi_{2dn^c}(D_{\text{start}}, D_{\text{accept}})$ to obtain a quantified boolean formula, and outputs it. $\square$

### 3.1.1 The essence of PSPACE: optimum strategies for game-playing.

Recall that the central feature of NP-complete problems is that a yes answer has a short certificate. In fact, this was essentially the definition of NP. The analogous unifying concept for PSPACE-complete problems seems to be that of a winning strategy for a 2-player game with perfect information. A good example of such a game is Chess: two players alternately make moves, and the moves are made on a board visible to both. Thus moves have no hidden sideeffects; hence the term “perfect information.” Finding whether or not the first player has a winning strategy involves searching the tree of all possible moves. (In particular, even describing the strategy takes exponentially many bits.) Note that the first player has a winning strategy iff there is a 1st move for player 1 such that for every possible 1st move of player 2 there is a 2nd move of player 1 such that.... (and so on) such that at the end player 1 wins. The interplay of existential and universal quantifiers in the previous line suggests the following example of a game.

**Example 3.4 (The QBF Game)** The “board” for the QBF game is a boolean formula $\varphi$ whose free variables are $x_1, x_2, \ldots, x_{2n}$. The two players alternately make moves, which involve picking values for $x_1, x_2, \ldots, $ in order. Thus player 1 will pick values for the odd-numbered variables $x_1, x_3, x_5, \ldots$ (in that order) and player 2 will pick values for the even-numbered variables $x_2, x_4, x_6, \ldots$. We say player 1 wins iff at the end $\varphi$ becomes true.
Clearly, player 1 has a winning strategy iff
\[ \exists x_1 \forall x_2 \exists x_3 \forall x_4 \cdots \forall x_{2n} \varphi(x_1, x_2, \ldots, x_{2n}), \]
namely, iff this quantified boolean formula is true.

Thus deciding whether player 1 has a winning strategy in the QBF game is \( \text{PSPACE-complete} \).

At this point, the reader is probably thinking of familiar games such as Chess, Go, Checkers etc. and wondering whether complexity theory may help differentiate between them—for example, to justify the common intuition that Go is more difficult than Chess. Unfortunately, formalizing these issues in terms of asymptotic complexity seems impossible because these are finite games, and in terms of the existence of a winning strategy, there are at most three choices: Player 1 has a winning strategy, Player 2 does, or neither does (they can play to a draw). However, one can study generalizations of these games to an \( n \times n \) board where \( n \) is arbitrarily large—this may involve stretching the rules of the game since the definition of chess seems tailored to an 8 \( \times \) 8 board—and then complexity theory can indeed be applied. For most common games, including chess, determining which player has a winning strategy in the \( n \times n \) version is \( \text{PSPACE-complete} \) (see Papadimitriou or Garey and Johnson). Note that if \( \text{NP} \neq \text{PSPACE} \) then there is also no short certificate for exhibiting that either player has a winning strategy, which is alluded to in Evens and Tarjan’s quote at the start of the chapter.

Proving \( \text{PSPACE-completeness} \) of games may seem like a frivolous pursuit, but similar ideas lead to \( \text{PSPACE-completeness} \) of some practical problems. Usually, these involve repeated moves that are in turn counteracted by an adversary. For instance, many computational problems of robotics are \( \text{PSPACE-complete} \): the “player” is the robot and the “adversary” is the environment. (Treating the environment as an adversary may appear unduly pessimistic; but unfortunately even assuming a benign or “indifferent” environment still leaves us with a \( \text{PSPACE-complete} \) problem; see the Chapter notes.)

### 3.2 NL completeness

Now we consider problems that form the “essence” of NL, in other words, problems that are \( \text{complete} \) for NL. What kind of reduction should we use?
3.2. NL COMPLETENESS

We cannot use the polynomial-time reductions since NL ⊆ P. Thus every language in NL is polynomial-time reducible to the trivial language \{1\} (reduction: “decide using polynomial time whether or not the input is in the NL language, and then map to 1 or 0 accordingly”). Clearly, such trivial languages should not be complete under any useful definition of reduction.

When choosing the type of reduction to define completeness for a complexity class, we must keep in mind the complexity phenomenon we seek to understand. In this case, the complexity question is whether or not NL = L. The appropriate reduction here happens to be logspace reductions (for justification, see part (b) of Lemma 3.3 below).

A logspace transducer is a deterministic machine that has three tapes: a read-only tape containing an input, a read-write work tape whose length is \(O(\log n)\) when the input has \(n\) bits, and a write-only output tape. For any input, the machine computes in \(\text{poly}(n)\) time (remember, a logspace machine cannot take any more than \(\text{poly}(n)\) time if it has to avoid looping forever) and during its computation it writes a string on its output tape. Since the output tape is write-only, the machine must start writing from the left end and move its head one cell to the right each time it writes a symbol. Thus such a transducer computes a function \(f : \{0, 1\}^* \rightarrow \{0, 1\}^*\).

**Definition 3.4** Language \(A\) is logspace reducible to language \(B\), denoted \(A \leq_l B\), if there is a function \(f : \{0, 1\}^* \rightarrow \{0, 1\}^*\) computable by a logspace transducer such that for every string \(x \in \{0, 1\}^*\), \(x \in A\) iff \(f(x) \in B\).

Logspace reducibility satisfies usual properties one expects.

**Lemma 3.3**

(a) If \(A \leq_l B\) and \(B \leq_l C\) then \(A \leq_l C\). (b) If \(A \leq_l B\) and \(B \in L\) then \(A \in L\).

**Proof:** We only prove part (a); part (b) is similar. Suppose the logspace reductions from \(A\) to \(B\) and from \(B\) to \(C\) are given by \(f_1, f_2 : \{0, 1\}^* \rightarrow \{0, 1\}^*\) respectively, and they are computed by transducers \(M_1, M_2\) respectively. The idea of course is that we want to reduce \(A\) to \(C\) by mapping string \(x\) to \(f_2(f_1(x))\): in other words, first compute string \(f_1(x)\) and then apply \(f_2\) to it. Clearly, \(f_2(f_1(x)) \in C\) iff \(x \in A\). The catch is that \(f_1(x)\) may have size polynomial in \(|x|\), and the logspace transducer we are trying to construct does not have enough space to store it. Thus the correct solution is to compute each bit of \(f_1(x)\) “on the fly” whenever it is needed by \(M_2\). In other words, our transducer simulates \(M_2\)’s actions if it were given the string \(f_1(x)\) on a “ghost” tape. Our transducer keeps track of the position of \(M_2\)’s head on this “ghost” tape.
CHAPTER 3. SPACE COMPLEXITY

Figure 3.3: Composing two logspace reductions. The “ghost” tape serves both as the output tape of $M_1$ and the input tape of $M_2$.

Whenever $M_2$ moves its head one cell to the left or right, our transducer starts simulating $M_1$ on input $x$, discarding all symbols produced by $M_1$ until it outputs the symbol in the position where $M_2$’s head now rests.

To be sure, the simulation is very slow, since each step of $M_2$ may require having to run $M_1$ more or less completely. But it runs in logspace. □

**Definition 3.5** Language $A$ is *NL-complete* if it is in NL and every NL language is logspace reducible to it.

Note that an NL-complete language is in L iff NL = L.

**Theorem 3.4**
PATH is NL-complete.

**Proof:** We have already seen that PATH is in NL. Let $L$ be any language in NL and $M$ be a machine that decides it in space $O(\log n)$. We describe a logspace transducer that reduces $L$ to PATH.

For any input $x$ of size $n$, the transducer outputs a graph $G_{M,x}$ which is the configuration graph of $M$ on $x$. This is a directed graph whose nodes are all possible $2^{O(\log n)}$ configurations of the machine on input $x$, and there is an edge from configuration $S_i$ to configuration $S_j$ iff the machine can go from $S_i$ to $S_j$ in one step. In this graph there is a path from the start configuration to the accepting configuration (assume there is a unique accepting configuration, by modifying the machine so that it cleans up its work tape before entering the accept state) iff $M$ accepts $x$.

The transducer outputs the set of configurations one by one, and for each configuration it outputs all edges going out of that configuration (the
3.3. TWO SUPRISING ALGORITHMS

number of such edges is some constant depending upon $M$. Clearly, this is possible in $O(\log n)$ space. □

3.3 Two suprising algorithms

Now we describe two surprising algorithms for space-bounded computation. They both concern the language PATH, which we saw is in NL. Note that PATH also has a trivial polynomial time (actually, near-linear time) algorithm that uses depth-first search; however, this algorithm uses linear space. Can we decide PATH deterministically in sublinear space? This seems difficult, since intuition says that a deterministic search algorithm must store information about every node it has visited. The next theorem is therefore surprising.

**Theorem 3.5 (Savitch [Sav70])**

$\text{PATH} \in \text{SPACE}(\log^2 n)$.

**Proof:** Let $<G,s,t>$ be the input. We describe a recursive procedure $\text{reach?}(u,v,l)$ that returns “YES” if there is a path from $u$ to $v$ of length at most $l$ and “NO” otherwise. The procedure uses the observation that if there is a path from $u$ to $v$ of length at most $l$, there is a “middle” node $z$ along this path such that there is a path of length $\lceil l/2 \rceil$ from $u$ to $z$ and $\lfloor l/2 \rfloor$ from $z$ to $v$.

The procedure is as follows. If $l = 1$, return YES iff $(u,v)$ is an edge. Otherwise for each node $z$, run $\text{reach?}(u,z,\lceil l/2 \rceil)$ and $\text{reach?}(u,z,\lfloor l/2 \rfloor)$ and return YES if both return YES.

The main observation is that the second invocation of the $\text{reach}$ procedure can run in the same space used by the first one. Also, keeping track of the current value of $z$ takes only $O(\log n)$ space. Thus if $S(l)$ denotes the space requirement, we have

$$S(l) \leq O(\log n) + S(\lceil l/2 \rceil).$$

This yields $S(l) = O(\log n \log l)$.

The final answer is $\text{reach?}(s,t,n)$, so the space required is $S(n) = O(\log^2 n)$. □

**Remark 3.1** Note that Savitch’s algorithm does not run in $\text{poly}(n)$ time, however, and could take $2^{O(\log^2 n)} = n^{O(\log n)}$ time.
In fact, the algorithm for PATH actually gives a way to simulate all nondeterministic space-bounded computations on a deterministic machine with at most a quadratic blowup in space requirements.

**Corollary 3.6**

For every \( S(n) > \log n \), \( \text{NSPACE}(S(n)) \subseteq \text{SPACE}(S(n)^2) \).

**Proof:** Similar to the above proof, where the “graph” in question is the configuration graph of the nondeterministic machine (this was encountered earlier in the proof of Theorem 3.1). Let \( M \) be a nondeterministic machine running in space \( S(n) \) and \( x \) be an input of length \( n \). Then the configuration graph is a directed graph whose nodes are all possible \( 2^{O(S(n))} \) configurations of the machine on input \( x \), and there is an edge from configuration \( C_1 \) to configuration \( C_2 \) iff the machine can go from \( C_1 \) to \( C_2 \) in one step. In this graph there is a path from the start configuration to the accepting configuration (assume there is a unique accepting configuration, by modifying the machine so that it cleans up its work tape before entering the accept state) iff \( M \) accepts \( x \).

Note also that the algorithm in the proof of Theorem 3.5 requires access to the input graph only to verify for pairs of nodes \((u, v)\) whether \( u \to v \) is an edge. In the current situation \( u, v \) are configurations and the machine can check for itself whether there is an edge from one to the other in the configuration graph. \( \Box \)

**Remark 3.2** Corollary 3.6 implies that \( \text{PSPACE} = \text{NPSPACE} \).

The next result concerns \( \overline{\text{PATH}} \), the complement of PATH. A decision procedure for this language must accept when there is no path from \( s \) to \( t \) in the graph. It seemed “obvious” to researchers\(^2\) that \( \overline{\text{PATH}} \notin \text{NL} \), until the discovery of the following theorem in the 1980s proved them wrong. This is the second of the two surprising algorithms alluded to in the title of this section.

**Theorem 3.7** (Immerman-Szelepcsenyi)

\( \overline{\text{PATH}} \in \text{NL} \).

**Proof:** For notational ease we assume the nodes are numbered from 1 to \( n \) and \( s = 1, t = n \). We have to describe a nondeterministic computation

\(^2\)Though sometimes undergraduates mistakenly find it “obvious” that \( \overline{\text{PATH}} \in \text{NL} \): just take the NDTM for PATH and flip a yes answer to a no answer and vice versa. We pointed out in our discussion of \( \text{coNP} \) in Chapter 2 why this is fallacious.
3.3. TWO SURPRISING ALGORITHMS

running in $O(\log n)$ space that accepts iff there is no path from $s$ to $t$. Let us first consider an easier problem. Suppose somebody gives the machine a number $c$, which is exactly the number of nodes reachable from $s$. (This count is $\geq 1$, since $s$ is reachable from $s$.) Then a little thought suggests that the task becomes easier. The machine enumerates nodes one by one from 1 to $n - 1$ and tries to guess —using a nondeterministic walk—a path of length at most $n$ from $s$ to that node. It keeps a count of how many nodes it succeeded for. It accepts at the end iff it succeeds for $c$ nodes, since that means that $t$ is not one of the $c$ nodes reachable from $s$. Note on the other hand that if every branch of this nondeterministic computation fails (i.e., does not accept) then there do not exist $c$ nodes different from $t$ that are reachable from $s$, which means that then $t$ must be reachable from $s$.

To give an NL algorithm for $\text{PATH}$, we show how to compute this magic number $c$ using nonderminism. (When we say that a nondeterministic computation “computes” a number, we require branches to either HALT without an answer, or to output the correct answer. Also, at least one branch has to output the correct answer.) We use an inductive counting technique, where step $i$ determines $c_i$, the number of nodes reachable from $s$ in $i$ steps. Thus $c_n$ is the same as $c$ in the above description.

Let $C_i$ be the set of nodes reachable from $s$ using a path of length at most $i$. Thus $c_i = |C_i|$.

\[ C_{i+1} = C_i \cup \{u \mid \text{there is a walk of length } i + 1 \text{ from } s \text{ to } u\}. \]

Figure 3.4: A node $u$ is in $C_{i+1} \setminus C_i$ iff it is not in $C_i$ and there is a walk of length $i + 1$ from $s$ to $u$. 
NONDETERMINISTIC PROCEDURE TO COMPUTE $c_{i+1} - c_i$
GIVEN $c_i$.
Maintain a counter called $\text{count}$ that is initialized to 0.
Do the following for each node $u$ from 1 to $n$.

1. *(Verify that $u \not\in C_i$;)*
   Enumerate all nodes from 1 to $n$ except $u$ and try to
guess a nondeterministic walk —whose length is also
nondeterministically picked from 1 to $i$— from $s$ to that
node of length at most $i$. Keep track of the number of
nodes for which this succeeds, and if this number equals
$c_i$, then we have managed to verify that $u \not\in C_i$. If so,
continue, else HALT and reject.

2. *(Verify that $u \in C_{i+1} \setminus C_i$)*
   Take a nondeterministic walk of length $i + 1$ from $s$. If
the walk does not end at $u$, then HALT.
   If the walk ends at $u$, then clearly $u \in C_{i+1}$. Since
we already verified that $u \not\in C_i$, we now conclude $u \in
C_{i+1} \setminus C_i$ and increment $\text{count}$ by 1.

Clearly, if the machine has not yet halted, then $\text{count}$ con-
tains $c_{i+1} - c_i$ at the end. Also, it is clear from the description
that only $O(\log n)$ bits of storage is needed.

Using the notion of the configuration graph as in Corollary 3.6 we can
modify the proof of Theorem 3.7 to prove the following.

**Corollary 3.8**
For every $s(n) > \log \log n \quad \text{NSPACE}(s(n)) = \text{coNSPACE}(s(n))$.

**Our understanding of space-bounded complexity.**
The following is our understanding of space-bounded complexity.

$$\text{DTIME}(s(n)) \subseteq \text{SPACE}(s(n)) \subseteq \text{NSPACE}(s(n)) = \text{coNSPACE}(s(n)) \subseteq \text{DTIME}(2^O(s(n))).$$

None of the inclusions are known to be strict though we believe they are.

**Exercises**

§1 Show that $\text{SPACE}(s(n)) = \text{SPACE}(0)$ when $s(n) = \log \log n$. 
3.3. TWO SURPRISING ALGORITHMS

§2 In analogy with the characterization of NP in terms of certificates, show that we can define NSPACE(s(n)) as the set of languages for which certificates can be checked deterministically using \( O(S(n)) \) space, where the certificate is provided on a read-only tape of size \( 2^{O(S(n))} \) and the verifier can scan it only once from left to right. (b) If \( A \leq_l B \) and \( B \in \mathbb{L} \) then \( A \in \mathbb{L} \).

§3 Show that the following language is NL-complete:

\[ \{ <G>: G \text{ is a strongly connected digraph} \} \]

§4 Show that 2SAT is in NL.

§5 Suppose we define NP-completeness using logspace reductions instead of polynomial-time reductions. Show (using the proof of the Cook-Levin Theorem) that SAT and 3SAT continue to be NP-complete under this new definition. Conclude that SAT \( \in \mathbb{L} \) iff NP = \( \mathbb{L} \).

§6 Show that TQBF is complete for PSPACE also under logspace reductions.

§7 Show that in every finite 2-person game with perfect information (by finite we mean that there is an a priori upperbound \( n \) on the number of moves after which the game is over and one of the two players is declared the victor —there are no draws) one of the two players has a winning strategy.

§8 Define polyL to be \( \bigcup_{c>0} \text{SPACE}(\log^c n) \). Steve’s Class SC (named in honor of Steve Cook) is defined to be the set of languages that can be decided by deterministic machines that run in polynomial time and \( \log^c n \) space for some \( c > 0 \).

It is an open problem whether PATH \( \in \mathbb{C} \). Why does Savitch’s Theorem not resolve this question?

Is SC the same as polyL \( \cap \mathbb{P} \)?

Chapter notes

The concept of space complexity had already been explored in the 1960s; in particular, Savitch’s theorem predates the Cook-Levin theorem. Stockmeyer and Meyer proved the PSPACE-completeness of TQBF soon after Cook’s paper appeared. A few years later Even and Tarjan pointed out
the connection to game-playing and proved the PSPACE-completeness of a game called Generalized Hex. Papadimitriou’s book gives a detailed account of PSPACE-completeness. He also shows PSPACE-completeness of several *Games against nature* first defined in [Pap85]. Unlike the TQBF game, where one player is *Existential* and the other *Universal*, here the second player chooses moves randomly. The intention is to model games played against nature—where “nature” could mean not just weather for example, but also large systems such as the stock market that are presumably “indifferent” to the fate of individuals. Papadimitriou gives an alternative characterization PSPACE using such games. A stronger result, namely, a characterization of PSPACE using interactive proofs, is described in Chapter 10.
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. [BGS75]

To separate two complexity classes we need to exhibit a machine in one class that is different (namely, 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. 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 undecidability of the Halting Problem.

Though diagonalization led to some 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 4.4. Thus diagonalization went out of favor for many years. But some recent results (see Section 18.5 for an example) use diagonalization as a key component. Thus future complexity theorists should master this simple idea before going on to anything fancier!
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. We say that a function \( f : \{0,1\}^* \to \mathbb{N} \) is a time-constructible function if there is a Turing machine that, given an input with \( n \) bits, writes down \( f(n) \) on its tape in \( f(n) \) 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

\[
\text{DTIME}(f(n)) \subsetneq \text{DTIME}(g(n)) \tag{4.1}
\]

To showcase the essential idea of the proof of Theorem 4.1, we prove the simpler statement \( \text{DTIME}(n) \subsetneq \text{DTIME}(n^2) \).

We use diagonalization. Suppose \( M_1, M_2, M_3, \ldots \) is a numbering of all Turing Machines, where the description of \( M_i \) can be produced from \( i \) in time \( O(i) \). (Such numberings exist. For example, one can order TMs according to the number of states in their transition diagrams, and use lexicographic ordering among then all TMs that have the same number of states. Note that we allow the numbering to include machines that may not halt on all inputs.)

Consider the following Turing Machine, \( D \): “On input \( x \), if \( x \not\in \{0,1\}^* \) then accept. If \( x = 0^j1^k \) for some \( j, k \) then construct \( M_k \) and simulate it on \( x \) for \( |x|^{1.5} \) steps. If \( M_k \) halts and accepts during this simulation, reject. If \( M_k \) halts and rejects, accept. If \( M_k \) does not halt in this time, accept.”

It may appear that this machine has running time \( O(|x|^{1.5}) \) but actually there is an overhead involved in simulating another machine (namely, to keep track of the number of steps using a counter). This overhead is at most a logarithmic factor (exercise), so the running time on \( x \) is at most \( O(|x|^{1.5} \log |x|) \). We can safely say that asymptotically speaking, machine \( D \) runs in at most \( 2n^2 \) time. Thus \( L(D) \), the language decided by \( D \), is in \( \text{DTIME}(n^2) \). The next claim implies that \( L(D) \not\in \text{DTIME}(n) \).

**Claim 4.2**

For every \( k \), if machine \( M_k \) runs in linear time, then there is a \( j > 0 \) such that on input \( 0^j1^k \), machines \( D \) and \( M_k \) give different answers.

**Proof:** Say the running time of \( M_k \) at most \( cn + d \) for some constants \( c,d \). If \( j + k > \max \{(c+1)^2,d\} \), then \( |y|^{1.5} > c|y| + d \), and so \( D \) on input
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.

**Theorem 4.3**

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

\[
\text{SPACE}(f(n)) \subsetneq \text{SPACE}(g(n)) \quad (4.2)
\]

4.3 Nondeterministic Time Hierarchy Theorem

The analogous hierarchy theorem for nondeterministic computation is even tighter than for deterministic computation: it drops the extra \( \log \) term. \( \text{TBE} \) later: why no extra \( \log \) term?

**Theorem 4.4**

If \( f, g \) are running times satisfying \( f(n) = o(g(n)) \), then

\[
\text{NTIME}(f(n)) \subsetneq \text{NTIME}(g(n)) \quad (4.3)
\]

Again, we just showcase the main idea of the proof by proving \( \text{NTIME}(n) \subsetneq \text{NTIME}(n^2) \). The technique from the previous section does not directly apply. A nondeterministic machine that runs in \( O(n) \) time may have \( 2^{O(n)} \) branches in its computation. It is unclear how to determine in \( O(n^2) \) time whether or not some branch accepts and then flip this answer. Instead we use a technique called lazy diagonalization, which is only guaranteed to flip the answer on some input in a fairly large range.

First, we need an enumeration of all NDTMs \( M_1, M_2, \ldots \); this is analogous to the enumeration of deterministic TMs. Furthermore, to simplify notation, we assume that every machine appears an infinite number of times in our enumeration. This could be ensured by allowing extraneous “padding” states in the finite control that are not reachable from the other states; these do not affect the machine’s operation but change its formal description.
Now consider the increasing sequence of numbers \( f(1), f(2), f(3), \ldots \), where
\[
f(i) = 2^{2^{2^{i}}}. \tag{4.4}
\]
It is easily checked that \( f(i + 1) \geq 2^{f(i)^2} \). Our diagonalizing machine will try to flip the answer of \( M_i \) on some input in the set \( \{1^n : f(i) < n \leq f(i + 1)\} \).

We note that given \( n \), it is easy to find in say \( n^2 \) time the \( i \) such that \( n \) is sandwiched between \( f(i) \) and \( f(i + 1) \). The diagonalizer NDTM \( D_1 \) is as follows.

"On input \( x \), if \( x \not\in \{1^n : f(i) < n \leq f(i + 1)\} \), accept. If \( x = 1^n \), then compute \( i \) such that \( f(i) < n < f(i + 1) \).

1. If \( f(i) < n < f(i + 1) \) then simulate \( M_i \) on input \( 1^{n+1} \) using nondeterminism in \( n^{1.5} \) time and output its answer.

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

Figure 4.1: When \( n \) lies inside the interval \( (f(i), f(i + 1)) \), the diagonalizer makes its decision on \( 1^n \) by simulating \( M_i \) on \( 1^{n+1} \) and using its answer.

Note that part (2) requires going through all possible \( \exp(|f(i)|^{1.5}) \) branches of \( M_i \) on input \( 1^{f(i)+1} \), but that is fine since the input size \( f(i + 1) \) exceeds even \( 2^{f(i)^2} \). We conclude that NDTM \( D_1 \) runs in \( O(n^2) \) time. The next claim shows that \( L(D_1) \not\in \NTIME(n) \).

**Claim 4.5**

For every language \( A \in \NTIME(n) \) there is an \( i \) such that \( L(M_i) = A \) but for some \( n \in (f(i), f(i + 1)] \), exactly one of \( M_i \) and \( D_1 \) accepts \( 1^n \).

**Proof:** Suppose \( A \) is decidable by an NDTM that runs in \( cn + d \) time. Since this NDTM appears infinitely often in the enumeration, it appears as \( M_i \) for some \( i > c + d \). Note that on inputs of length more than \( f(i) \), machine \( M_i \) runs in at most \( n^{1.5} \) time. Thus the two steps in the description of \( D_1 \) ensure respectively that

\[
\text{If } f(i) < n < f(i + 1), \text{ then } D_1 \text{ accepts } 1^n \text{ iff } M_i \text{ accepts } 1^{n+1}. \tag{4.5}
\]
\[
D_1 \text{ accepts } 1^{f(i)+1} \text{ iff } M_i \text{ does not accept } 1^{f(i)+1}. \tag{4.6}
\]
Now suppose for contradiction’s sake, $M_i$ and $D_1$ agree on all inputs $1^n$ where $f(i) < n \leq f(i+1)$. Then together with (4.5) this would imply that $D_1$ accepts $1^{f(i+1)}$ iff $M_i$ accepts $1^{f(i)+1}$, which contradicts (4.6).

4.4 Can diagonalization resolve P vs NP?

Quantifying the limits of “diagonalization” is not easy. Certainly, the diagonalization in Section 4.3 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 enumeration of all Turing machines $M_1, M_2, \ldots$,

2. The ability of one TM to simulate any another without much overhead in running time or space. This is a stronger version of the classical theorem about the existence of a universal Turing machine: we are asserting that the universal Turing machine, when given the index $i$ and some input $x$, can simulate machine $M_i$ on $x$ without significant overhead.

Any argument that only uses these facts is treating machines as blackboxes: the machine’s internal workings do not matter. To formalize this further we need the following notion.

**Definition 4.1 (Oracle Turing Machines)** If $B$ is a language, then a machine $M$ with access to oracle $B$, denoted $M^B$, is a machine with a special query tape. It can write down any string on this tape, and learn from the oracle in a single step whether or not the string is in the language $B$.

We denote by $P^B$ the class of languages accepted by deterministic polynomial time machines that have access to oracle $B$. We similarly define $C^B$ for any complexity class $C$.

**Example 4.1** $\text{SAT} \in P^{\text{SAT}}$. To decide whether a formula $\varphi \in \text{SAT}$, the machine asks the oracle if $\varphi \in \text{SAT}$, and then gives the opposite answer as its output.
In general, asking for an oracle $C$ whether or not $P^C = NP^C$ is called \textit{relativization} of the $P$ versus $NP$ question \textit{with respect to} the oracle $C$.

Note that the two properties needed for diagonalization arguments continue to hold when all machines are provided with the same oracle. There is an effective enumeration of Turing machines since an oracle Turing machine’s description (minus the oracle) is of finite size. Furthermore, the Universal machine can still simulate any other machine — whenever the machine being simulated queries the oracle, so does the Universal machine.

We conclude that if we could resolve $P$ vs $NP$ — in whichever direction — using a proof technique involving only simulation then the proof would also work in the presence of every oracle. Hence the following theorem implies that such a proof cannot exist.

**Theorem 4.6 (Baker, Gill, Solovay)**

There exist oracles $B, C$ such that $P^C = NP^C$ and $P^B \neq NP^B$.

**Proof:** For $C$ we may take any PSPACE-complete problem, say TQBF, since $P^{TQBF} = NP^{TQBF} = PSPACE$. (See exercises.) Now we construct $B$. For any language $A$, let $A_u$ be the unary language

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

For every oracle $A$, the language $A_u$ is clearly in $NP^A$. Below we construct an oracle $B$ such that $B_u \notin P^B$. Hence $B_u \in NP^B \setminus P^B$, and we conclude $P^B \neq NP^B$.

**Construction of B:** Let $M_1, M_2, M_3, \ldots$ be all polynomial-time Oracle Turing Machines. (This enumeration need not be effective, since we are merely showing the \textit{existence} of the desired oracle.) We construct $B$ in stages, where stage $i$ ensures that $M_i^B$ does not decide $B_u$. 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 $2^n$ exceeds the running time of $M_i$ on inputs of length $n$. Now run $M_i$ on input $1^n$. 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. We continue until $M_i$ halts. Now we make sure 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 $2^n$ exceeds the running time of $M_i$) and declare that it is in $B$, thus ensuring $1^n \in B_u$. In either case, the answer of $M_i$ is incorrect.

Thus our construction of $B$ ensures that for every $i$, machine $M_i$ does not decide $B_u$. □

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 2: the Cook-Levin theorem! It is not true for a general oracle $A$ that every language $L \in \text{NP}^A$ is polynomial-time reducible to 3SAT (see exercises). Note however that nonrelativizing facts are necessary, not sufficient. It is an open question how to use known nonrelativizing facts in resolving $P$ vs $NP$ (or other 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 3.6) and Theorem 3.7 do relativize.

**Exercises**

§1 Show that the following language is undecidable:

\[
\{ \langle M \rangle : M \text{ is a machine that runs in } 100n^2 + 200 \text{ time} \}.
\]

§2 Show that maintaining a time counter can be done with logarithmic overhead. (Hint??)

§3 Show that $P^{\text{TQBF}} = \text{NP}^{\text{TQBF}}$.

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

§5 Show that there is a language $B \in \text{EXPTIME}$ such that $\text{NP}^B \neq P^B$.

§6 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 $\text{DTIME}(n^{1.1})$ superior to $\text{DTIME}(n)$?
(b) Is $\text{NTIME}(n^{1.1})$ superior to $\text{NTIME}(n)$?

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

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

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

§10 Note that if $\text{P} = \text{NP}$ then every language in $\text{P}$ is $\text{NP}$-complete. Prove the following statement (Ladner's theorem): If $\text{P} \neq \text{NP}$ then there is a language in $\text{NP}$ that is neither in $\text{P}$ nor $\text{NP}$-complete. (Hint: Take a language in $\text{P}$ and a language in $\text{NP} \setminus \text{P}$ and form a new language by interlacing the two. Note that you need to diagonalize against all polynomial-time reductions.)

Chapter notes and history

The time hierarchy theorem is from Hartmanis and Stearns' pioneering paper [HS65]. The space hierarchy theorem is from Stearns, Hartmanis, and Lewis [SHL65]. The nondeterministic time hierarchy theorem is from Cook [Coo73], though the simple proof given here is essentially from [Zak83]. A similar proof works for other complexity classes such as the (levels of the) polynomial hierarchy discussed in the next chapter. The notion of relativizations of the $\text{P}$ versus $\text{NP}$ question is from Baker, Gill, and Solovay [BGS75], though the authors of that paper note that other researchers independently discovered some of their ideas.

The notion of oracle Turing machines can be used to study interrelationships of complexity classes. In fact, Cook [Coo71] defined $\text{NP}$-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, to me 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.
CHAPTER 4. DIAGONALIZATION
Chapter 5

Alternating TM’s and Polynomial Hierarchy

“..synthesizing circuits is exceedingly difficulty. It is even more
difficult to show that a circuit found in this way is the most
economical one to realize a function. The difficulty springs from
the large number of essentially different networks available.”
Claude Shannon 1949

This chapter discusses the alternating Turing Machine (ATM), a general-
alization of the nondeterministic Turing machine. Recall that even though
a NDTM is not a realistic computational model, studying it helps us to
focus on a natural computational phenomenon, namely, the apparent difference between guessing an answer and verifying it. The ATM plays a similar role for certain languages for which there is no obvious short certificate for membership and hence cannot be characterized using nondeterminism alone. These machines will be used to define the polynomial hierarchy, a general-
ization of P, NP and coNP that tends to crop up in many complexity theoretic investigations (including many chapters of this book).

To understand the need for going beyond nondeterminism, let’s recall an NP problem, CLIQUE, for which we do have a short certificate of membership.

CLIQUE = \{ <G,k> : \text{graph } G \text{ has a clique of size } \geq k \} . \quad (5.1)

Consider a slight modification to the above problem, namely, determinin-
ing the largest clique in a graph (phrased as a decision problem):

\[ \text{EXACT-CLIQUE} = \{ < G, k > : \text{the largest clique in graph } G \text{ has size exactly } k \} \, . \tag{5.2} \]

Now there seems to be no short certificate for membership: \( < G, k > \in \text{EXACT-CLIQUE} \) iff there exists a clique of size \( k \) in \( G \) and every other clique has size at most \( k \).

Similarly, consider the language MIN-DNF, the decision version of a problem in circuit minimization, a topic of interest in electrical engineering (and referred to in Shannon’s paper). We say that two boolean formulae are equivalent if they have the same set of satisfying assignments.

\[ \text{MIN-DNF} = \{ < \varphi > : \varphi \text{ is a DNF formula not equivalent to any smaller DNF formula} \} \, . \tag{5.3} \]

\[ = \{ < \varphi > : \forall \psi, |\psi| < |\varphi|, \exists \text{ assignment } s \text{ such that } \varphi(s) \neq \psi(s) \} \, . \tag{5.4} \]

Again, there is no obvious notion of a certificate of membership. Note that both the above problems are in PSPACE, but neither is believed to be PSPACE-complete.

Clearly, one is faced with a new kind of complexity phenomenon and it is instructive to see how we define the right complexity class, namely, the polynomial hierarchy, to “capture” it. We will define it in at least four ways in this book, and the first definition is a simple generalization of Definitions 2.2, 2.4.

**Definition 5.1 (Polynomial Hierarchy)** The polynomial hierarchy is defined as \( \cup_{i \geq 0} \Sigma^p_i \) where

1. \( \Sigma^p_0 = \Pi^p_0 = \mathbb{P} \).
2. \( \Sigma^p_i \) consists of any language \( L \) for which there is a language \( L_0 \in \Pi^p_{i-1} \) and \( c, d > 0 \) such that

\[ x \in L \iff \exists y \in \{0, 1\}^*, |y| \leq d|x|^c \text{ and } (x, y) \in L_0. \]

3. \( \Pi^p_i \) consists of any language \( L \) for which there is a language \( L_0 \in \Sigma^p_{i-1} \) such that

\[ x \in L \iff \forall y \in \{0, 1\}^*, |y| \leq d|x|^c \text{ and } (x, y) \in L_0. \]
Remark 5.1 In particular, $\Sigma_1^p = \text{NP}$, $\Pi_1^p = \text{coNP}$.

Example 5.1 To understand this definition, let us unwrap it for $i = 2$. Language $L$ is in $\Sigma_2^p$ if there is a language $L_2 \in \text{P}$ and constants $c, d > 0$ such that a string $x$ is in $L$ iff

$$\exists y_1 \leq d|x|^c \forall y_2 \leq d|x|^c \ (x, y_1, y_2) \in L_2.$$  

Similarly Language $L$ is in $\Pi_2^p$ if there is a language $L_2 \in \text{P}$ and constants $c, d > 0$ such that a string $x$ is in $L$ iff

$$\forall y_1 \leq d|x|^c \exists y_2 \leq d|x|^c \ (x, y_1, y_2) \in L_2.$$  

Clearly, $L \in \Pi_2^p$ iff $\overline{L} \in \Sigma_2^p$.

Similarly we can unwrap the definition for general $i$ and directly define $\Sigma_i^p$ using $i$ quantifiers, the first being $\exists$ and the rest alternating between $\exists$ and $\forall$. The class $\Pi_i^p$ involves $i$ quantifiers, alternating between $\exists$ and $\forall$ and beginning with $\forall$.

We note that $\Sigma_i^p \subseteq \Pi_{i+1}^p$, so we can also define the polynomial hierarchy as $\cup_{i \geq 0} \Pi_i^p$. Now we consider some natural problems in PH.

Example 5.2 We show that EXACT-CLIQUE $\in \Sigma_2^p \cap \Pi_2^p$. (A finer placement of EXACT-CLIQUE is done in the exercises.)

To see that EXACT-CLIQUE $\in \Sigma_2^p$, note that $< G, k > \in \text{EXACT-CLIQUE}$ iff

$$\exists S \forall S' \text{ set } S \text{ is a clique of size } k \text{ in } G \text{ and } S' \text{ is not a clique of size } \geq k + 1.$$  

Furthermore, EXACT-CLIQUE $\in \Pi_2^p$ since one could also say

$$\forall S' \exists S \text{ set } S' \text{ is not a clique of size } \geq k + 1 \text{ and } S \text{ is a clique of size } k \text{ in } G.$$  

The reader can similarly check that MIN-DNF in $\Pi_2^p$. It is conjectured to be complete for $\Pi_2^p$.

The next simple theorem is the only theorem in this chapter.
CHAPTER 5. ALTERNATING TM’S AND POLYNOMIAL HIERARCHY

Theorem 5.1

1. If $P = NP$ then $PH = P$.

2. If $\Sigma^p_i = \Pi^p_i$ then $PH = \Sigma^p_i$. ("$PH$ collapses to the $i$th level.")

Proof: We do the first part; the second part is similar and also easy.

Assuming $P = NP$ we prove by induction on $i$ that $\Sigma^p_i, \Pi^p_i \subseteq P$. Clearly this is true for $i = 0$. Assume it is true for $i \leq m$ and $L \in \Sigma^p_{m+1}$. Then by Definition 5.1 there is a language $L_0 \in \Pi^p_m$ and $c, d > 0$ such that

$$\forall x \in \{0,1\}^* \ x \in L \iff \exists y \in \{0,1\}^*, |y| \leq d|x|^c \text{ and } (x,y) \in L_0.$$ 

By the inductive hypothesis, $L_0 \in P$. By examining how $L$ is defined in terms of $L_0$, we conclude that then $L \in NP$. Hence $L$ must be in $P$, and the induction is completed. $\square$

We strongly believe that not only is $P \neq NP$ but also that all levels of $PH$ are distinct. This latter conjecture will be useful in the rest of the course; we will reduce other conjectures to it (in other words, show that the other conjectures follow from this conjecture).

5.0.1 Complete problems for levels of $PH$

The class $PH$ as a whole is believed to not have a complete problem. However, each level of $PH$ does have complete problems.

The class $\Sigma^p_i$ has a complete problem (proving this is left as a simple exercise) involving quantified boolean expression with limited number of alternations. Specifically, it is

$$\Sigma^i\text{-SAT} = \exists \vec{x}_1 \forall \vec{x}_2 \exists \cdots Q \vec{x}_i \ \varphi(\vec{x}_1, \vec{x}_2, \ldots, \vec{x}_i) = 1, \quad (5.5)$$

where $\varphi$ is a boolean formula, each $\vec{x}_i$ is a vector of boolean variables, and $Q$ is $\exists$ or $\forall$ depending on whether $i$ is odd or even. Notice that this is a subcase of TQBF defined in Chapter 3.

Now we describe a problem that was recently proved to be complete for $\Sigma^p_2$ and plays a useful role in further reductions. In the SUCCINCT SET COVER problem we are given a collection $S = \{\varphi_1, \varphi_2, \ldots, \varphi_m\}$ of 3-DNF formulae on $n$ variables, and an integer $k$. We need to determine whether there is a subset $S' \subseteq \{1,2,\ldots,m\}$ of size at most $K$ for which $\forall i \in S' \varphi_i$ is a tautology (i.e., evaluates to 1 for every assignment to the variables).
5.1 Alternating Turing machines

Now we give a different characterization of PH using alternating Turing machines. (Aside: A third characterization using oracle Turing machines is explored in the exercises. A fourth characterization using uniform circuits is described in Chapter 6.)

Alternating TMs are described similarly to nondeterministic TMs with the additional feature that every internal state except \( q_{\text{acc}} \) and \( q_{\text{halt}} \) is labelled with either \( \exists \) or \( \forall \). A nondeterministic TM is a special case in which states are labelled with only \( \exists \). Similar to the NDTM, an ATM can evolve at every step in two possible ways. Recall Definition 1.1 for when an NDTM accepts the input: the machine is said to “accept” iff there is a branch from the start configuration (the root of the tree) to a leaf where in the leaf configuration the machine is in state \( q_{\text{acc}} \). Thus the notion of when an input is “accepted” was defined using the entire tree of configurations. Something similar happens for the ATM as well.

![Figure 5.1: Each configuration of an ATM is labeled with a \( \forall \) or \( \exists \). A \( \forall \) configuration (resp., \( \exists \) configuration) is labeled accept iff both (resp., at least one) configurations it leads to are labeled accept.](image)

Configurations in which the machine is in \( q_{\text{acc}} \) are said to be \textit{accepting}, just as in the case of an NDTM. A configuration in which the machine’s state is labeled with \( \exists \) is said to be accepting if at least one of the two configurations leading out from it is labeled accepting. A configuration in which the machine’s state is labeled with \( \forall \) is said to be accepting if both configurations leading out from it are labeled accepting. Note that these rules are consistent with the obvious semantics for \( \exists \) and \( \forall \). These rules are used to assign \textit{accepting/not accepting} labels to all nodes in the tree. The machine is said to accept if the start configuration (the root) gets an \textit{accepting} label. Note that when all states are labeled only with \( \exists \), the definition yields our earlier notion of acceptance for an NDTM.

Then \( \Sigma_i^p \) is the class of languages accepted by polynomial time alter-
nating TMs in which each computation branches features at most \((i - 1)\) alternations between \(\exists\) and \(\forall\), and the machine always starts in an \(\exists\) state. (In other words, \(q_{\text{start}}\) is an \(\exists\) state.) The class \(\Pi_p^{i}\) is similarly defined except the machine always starts in a \(\forall\) state.

It is left as an exercise to check that the definition of \(\text{PH}\) using ATMs is identical to our earlier definitions.

5.1.1 Unlimited number of alternations?

What if we consider polynomial-time alternating Turing machines with no \(a\) \(\text{priori}\) bound on the number of quantifiers? The class of languages accepted by them is called \(\text{AP}\).

**Theorem 5.2**

\(\text{AP} = \text{PSPACE}\).

**Proof:** \(\text{PSPACE} \subseteq \text{AP}\) follows since \(\text{TQBF}\) is trivially in \(\text{AP}\) (just “guess” values for each existentially quantified variable using an \(\exists\) state and for universally quantified variables using a \(\forall\) state) and every \(\text{PSPACE}\) language reduces to \(\text{TQBF}\).

\(\text{AP} \subseteq \text{PSPACE}\) follows using a recursive procedure similar to the one used to show that \(\text{TQBF} \in \text{PSPACE}\). \(\square\)

Similarly, one can consider alternating Turing machines that run in polynomial space. The class of languages accepted by such machines is called \(\text{APSPACE}\), and the exercises ask you to prove that \(\text{APSPACE} = \text{EXPTIME}\). One can similarly consider alternating logspace machines; the set of languages accepted by them is exactly \(\text{P}\).

**Problems**

§1 Show that the language in (5.5) is complete for \(\Sigma_p^i\) under polynomial time reductions. (Hint use the \(\text{NP}\)-completeness of \(\text{SAT}\).)

§2 Show that if \(3\text{SAT}\) is polynomial-time reducible to \(\overline{3}\text{SAT}\) then \(\text{PH} = \text{NP}\).

§3 Show that \(\text{PH}\) has a complete language iff it collapses to some finite level \(\Sigma_p^i\).

§4 Show that the definition of \(\text{PH}\) using ATMs coincides with our other definitions.
§5 Show that $APSPACE = EXPTIME$. (Hint: The nontrivial direction $EXPTIME \subseteq APSPACE$ uses ideas similar to those in the Cook-Levin theorem, namely, that computation is very local. The computation of an $APSPACE$ machine should be very reminiscent of a game between an $\exists$ player and a $\forall$ player.)

§6 Show that $\Sigma_2^p = NP^{SAT}$. Generalize your proof to give a characterization of $PH$ in terms of oracle Turing machines.

§7 The class DP is defined as languages $L$ for which there are two languages $L_1 \in NP$, $L_2 \in coNP$ such that $L = L_1 \cap L_2$. (Remark: Do not confuse DP with $NP \cap coNP$, which may seem superficially similar.) Show that

(a) EXACT-CLIQUE $\in$ DP.

(b) Every language in DP is polynomial-time reducible to EXACT-CLIQUE.

§8 Suppose $A$ is some language such that $P^A = NP^A$. Then show that $PH^A \subseteq P^A$ (in other words, the proof of Theorem 5.1 relativizes).

§9 Show that SUCCINCT SET-COVER $\in \Sigma_2^p$.

§10 (suggested by C. Umans) This problem studies VC-dimension, a concept important in machine learning theory. If $S = \{S_1, S_2, \ldots, S_m\}$ is a collection of subsets of a finite set $U$, the VC dimension of $S$, denoted $VC(S)$, is the size of the largest set $X \subseteq U$ such that for every $X' \subseteq X$, there is an $i$ for which $S_i \cap X = X'$. (We say that $X$ is shattered by $S$.)

A boolean circuit $C$ succinctly represents collection $S$ if $S_i$ consists of exactly those elements $x \in U$ for which $C(i, x) = 1$. Finally, $VC$\{-dimension\} $= \{<C, k>: C$ represents a collection $S$ s.t. $VC(S) \geq k}\}.$

(a) Show that VC-DIMENSION $\in \Sigma_3^p$.

(b) Show that VC-DIMENSION is $\Sigma_3^p$-complete. (Hint: Reduce from $\Sigma_3$-3SAT. Also, the collection $S$ produced by your reduction can use the same set multiple times.)
Chapter notes and history

The polynomial hierarchy was formally defined by Stockmeyer [Sto77], though the concept appears in the literature even earlier. For instance, Karp [Kar72] notes that “a polynomial-bounded version of Kleene’s Arithmetic Hierarchy (Rogers 1967) becomes trivial if $P = NP$.”

The class $\text{DP}$ was defined by Papadimitriou and Yannakakis [PY82], who used it to characterize the complexity of identifying the facets of a polytope.

The class of complete problems for various levels of $\text{PH}$ is not as rich as it is for $\text{NP}$, but it does contain several interesting ones. See Schaeffer and Umans [SU02a, SU02b] for a recent list. The Succinct Set-Cover problem is from Umans [Uma01], where it is also shown that the following optimization version of MIN-DNF is $\Sigma^p_2$-complete:

\[
\{ \langle \varphi, k \rangle : \exists \text{ DNF} \varphi' \text{ of size at most } k, \text{ that is equivalent to DNF } \varphi \}.
\]
Chapter 6

Circuits

“One might imagine that $P \neq NP$, but SAT is tractable in the following sense: for every $l$ there is a very short program that runs in time $l^2$ and correctly treats all instances of size $l$. Theorem 5.2 shows that if “very short” means “of length $c \log l$” then this speculation is false.”

Karp and Lipton, 1982

The boolean circuit model is a formalization of the silicon chip. A circuit with $n$ inputs is a directed acyclic graph with $n$ leaves (called the input wires) and each internal node is labelled with $\lor, \land, \neg$ (in other words, the logical operations OR, AND, and NOT). The $\lor$ and $\land$ nodes have fanin (i.e., number of incoming edges) of 2 and the $\neg$ nodes have fanin 1. There is a designated output node. We think of $n$ bits being fed into the input wires. Each internal node/gate applies the boolean operation it is labeled with on its incoming wires to produce a single bit. Thus bits filter through the circuit —remember that it is acyclic, so feedback effects are absent— until a single output bit emerges at the output node. Thus the circuit implements a function from $\{0,1\}^n \rightarrow \{0,1\}$. The size of the circuit is the number of nodes in it.

We recall that the boolean operations OR, AND, and NOT form a universal basis, by which we mean that every boolean function on $n$-bit inputs can be implemented by a circuit of the type described above. Furthermore, OR, AND, and NOT gates are easily implemented in VLSI (also known as a “silicon chip”). A small circuit may be thus seen as an efficient computational model. It suffers from a restriction not shared by a TM, namely, it can only handle inputs of a certain size ($n$-bit strings in the above description). Nevertheless, one should consider the possibility that 3SAT instances
CHAPTER 6. CIRCUITS

Figure 6.1: A circuit with 2 inputs. It computes the XOR of the two inputs.

of up to say 10,000 variables could be solved by a very small circuit. If so, one could imagine a government-financed project to discover such a small circuit, which would then be implementated as a silicon chip and used in all kinds of commercial products. In particular, such a chip would jeopardise all current cryptographic schemes. This is the kind of speculation Karp and Lipton are referring to above, and will be one of the topics in this chapter.

To study the complexity of deciding a language by circuits, we again resort to asymptotic analysis. A circuit family \((W_n)_{n \geq 1}\) is a sequence of circuits where \(W_n\) is a circuit with \(n\) inputs. We say that the family decides a language \(L\) if for every input \(x\),

\[ x \in L \iff W_{|x|}(x) = 1. \quad (6.1) \]

Note that every language is decidable by a circuit family of size \(O(n^{2^n})\), since the circuit for input length \(n\) could contain \(2^n\) “hardwired” bits indicating which inputs are in the language. Given an input, the circuit looks up the answer from this table. (The reader may wish to work out an implementation of this circuit.) The following definition formalizes what we can think of as “small” circuits.

**Definition 6.1** The class \(P/\text{poly}\) consists of every language that is decidable by a circuit family of size \(O(n^c)\) for some \(c > 0\).

Of course, one can make the same kind of objections to the practicality of \(P/\text{poly}\) that one could make for \(P\): viz., in what sense is a circuit family of size \(n^{100}\) practical, even though it has polynomial size. This was answered to some extent in Section 2.0.2. Another answer is that as complexity theorists we hope (eventually) to show that languages such as SAT are not in \(P/\text{poly}\). Thus the result will only be stronger if we allow even such large circuits in the definition of \(P/\text{poly}\).
The next theorem shows that $\mathsf{P/poly}$ contains $\mathsf{P}$. The proof will follow from an even stronger Theorem 6.2.

**Theorem 6.1**

$\mathsf{P} \subseteq \mathsf{P/poly}$.

Can we give a good upperbound on the computational power represented by $\mathsf{P/poly}$? The next example shows this is difficult since $\mathsf{P/poly}$ contains an undecidable language.

**Example 6.1**

Every unary language has linear size circuits since the circuit for an input size $n$ only needs to have a single “hardwired” bit indicating whether or not $1^n$ is in the language. Hence the following unary language has linear size circuits, even though it is undecidable:

$$\{1^{<M,w>} : \text{TM } M \text{ accepts } w\}, \quad (6.2)$$

where $<M,w>$ is an encoding that encodes strings with integers.

This example suggests that it may be fruitful to only consider circuits that can actually be built, say using a fairly efficient Turing machine. This motivates the following definition.

**Definition 6.2 (logspace-uniform)** A circuit family $\{C_n\}$ is logspace uniform if there is a logspace transducer which, given the input $1^n$, outputs a description of $C_n$.

However, polynomial circuits that are logspace-uniform correspond to a familiar complexity class.

**Theorem 6.2**

A language has logspace-uniform circuits of polynomial size iff it is in $\mathsf{P}$.

**Proof:** The only if part is trivial. We prove the if part.

We show that every language that is decidable in $t(n)$ time has logspace-uniform circuits of size $O(t(n)^2)$. (A more careful argument can actually yield circuits of size $O(t(n) \log t(n))$. The main idea is similar to that in the Cook-Levin theorem, where we noticed that computation is very local, since the machine affects its tape only in the cell currently occupied by its head.

On any input of size $n$, the tableau of the machine’s computation is a matrix of size $t(n) \times t(n)$. Assume wlog that the machine enters the leftmost
cell before accepting. Construct a circuit in which use a wire for each bit of
the tableau. The bits of the machine’s input enter at the input wires of the
circuit. Construct a “next step” module, which is a circuit that, given the
contents of three adjacent cells in a row of the tableau, computes the contents
of the cell below the middle cell. Since the machine has a fixed transition
diagram and alphabet, each tableau cell is represented by $k = O(1)$ bits, so
the next-step circuit has size $O(1)$ (could be exponential in $k$). Replicate
this next-step module everywhere as needed so that all the bits in the tableau
are computed correctly. (See Figure 6.2.) The output of the overall circuit
is computed by examining the cell in the lower left corner of tableau; it is 1
iff the machine is in an accept state.

Clearly, the description of this circuit is very local, and a logspace ma-
chine could output the description in a gate-by-gate fashion. ☐

6.0.2 Turing machines that take advice

There is a way to define $\mathsf{P/poly}$ using TMs. We say that a Turing machine
has access to an advice family $\{a_n\}_{n \geq 0}$ (where each $a_n$ is a string) if while
computing on an input of size $n$, the machine is allowed to examine $a_n$. The
advice family is said to have polynomial size if there is a $c \geq 0$ such that
$|a_n| \leq n^c$.

Example 6.2 We showed that the language in (6.2) has linear size circuits.
Now we show that can be decided by a polynomial time Turing machine with 1 bit of advice. The advice string for inputs of length \( n \) is the single bit indicating whether or not \( 1^n \) is in the language.

The previous example is an example of a more general phenomenon described in the next theorem.

**Theorem 6.3**

\( L \in P/\text{poly} \) iff \( L \) is decidable by a polynomial-time Turing machine with access to an advice family of polynomial size.

**Proof:** If \( L \in P/\text{poly} \), we can provide the description of its circuit family as advice to a Turing machine. When faced with an input of size \( n \), the machine just simulates the circuit for this circuit provided to it.

Conversely, if \( L \) is decidable by a polynomial-time Turing machine with access to an advice family of polynomial size, then we can use the construction of Theorem 6.1 to construct an equivalent circuit for each input size with the corresponding advice string hardwired into it. \( \Box \)

### 6.1 Karp-Lipton Theorem

Karp and Lipton formalized the question of whether or not SAT has small circuits as: Is SAT in \( P/\text{poly} \)? They showed that the answer is “No” if the polynomial hierarchy does not collapse.

**Theorem 6.4 (Karp-Lipton, with improvements by Sipser)**

If \( NP \subseteq P/\text{poly} \) then \( PH = \Sigma^p_2 \).

Note that if \( NP \subseteq P/\text{poly} \) then SAT has a polynomial-size circuit family. Namely, for some \( k > 0 \) there is a sequence of circuits \( W_1, W_2, W_3, \ldots \), where \( W_n \) is a circuit of size \( n^k \) that decides satisfiability of all boolean formulas whose encoding size is \( \leq n \). (As per our definition, \( W_n \) only needs to solve instances of size exactly \( n \), but instances of size \( < n \) can be viewed as instances of size exactly \( n \) after adding some useless clauses like \( x_i \lor \overline{x_i} \).) Of course, \( NP \subseteq P/\text{poly} \) merely implies the existence of such a family; there may be no easy way to construct the circuits.

**Lemma 6.5 (Self-reducibility of SAT)**

There is a polynomial-time computable function \( h \) such that if \( \{W_n\}_{n \geq 1} \) is a circuit family that solves SAT, then for all boolean formulae \( \varphi \):

\[
\varphi \in SAT \quad \text{iff} \quad h(\varphi, W_{|\varphi|}) \text{ is a satisfying assignment for } \varphi. \quad (6.3)
\]
Proof: The main idea in the computation of $h$ is to use the provided circuit to generate a satisfying assignment. Ask the circuit if $\varphi$ is satisfiable. If so, ask it if $\varphi(x_1 = T)$ (i.e., $\varphi$ with the first variable assigned True) is satisfiable. The circuit’s answer allows us to reduce the size of the formula. If the circuit says no, we can conclude that $\varphi(x_1 = F)$ is true, and have thus reduced the number of variables by 1. If the circuit says yes, we can substitute $x_1 = T$ and again reduced the number of variables by 1. Continuing this way, we can generate a satisfying assignment. This proves the Lemma. (Aside: The formal name for the above property of SAT is downward self-reducibility. All NP-complete languages have this property.) □

Now we prove the Theorem.

Proof: (Theorem 6.4) We show $\text{NP} \subseteq \text{P/poly}$ implies $\Pi_2^p \subseteq \Sigma_2^p$.

Let $L \in \Pi_2^p$. Then there is a language $L_1 \in \text{NP}$ and $c > 0$ such that

$$L = \{ x \in \{0, 1\}^\ast : \forall y, |y| \leq |x|^c, (x, y) \in L_1 \}.$$  \hspace{1cm} (6.4)

Since $L_1 \in \text{NP}$, there is a polynomial-time reduction, say $g$, from $L_1$ to SAT. Thus

$$\forall z \in \{0, 1\}^\ast : z \in L_1 \iff g(z) \in \text{SAT}.$$  

Suppose further that $d > 0$ is such that $|g(z)| \leq |z|^d$.

Now we can rewrite (18.14) as

$$L = \{ x \in \{0, 1\}^\ast : \forall y, |y| \leq |x|^c, g(x, y) \in \text{SAT} \}.$$  

Note that $|g(x, y)| \leq (|x| + |y|)^d$. Let us simplify this as $|x|^{cd}$. Thus if $W_{n \geq 1}$ is a $n^k$-sized circuit family for SAT, then by (18.13) we have

$$L = \{ x : \forall y, |y| \leq |x|^c, h(g(x, y), W_{|x|^c}) \text{ is a satisfying assignment for } g(x, y) \}.$$  

Now we are almost done. Even though there may be no way for us to construct the circuit for SAT, we can just try to “guess” it. Namely, an input $x$ is in $L$ iff

$$\exists W, \text{ a circuit with } |x|^{cd} \text{ inputs and size } |x|^{cdk} \text{ such that}$$

$$\forall y, |y| \leq |x|^c, h(g(x, y), W) \text{ is a satisfying assignment for } g(x, y).$$

Since $h$ is computable in polynomial time, we have thus shown $L \in \Sigma_2^p$. □

Similarly the following theorem can be proven, though we leave the proof as an exercise.
6.2. CIRCUIT LOWERBOUNDS

Theorem 6.6 (Karp-Lipton, attributed to A. Meyer)
If $\text{EXPTIME} \subseteq P/\text{poly}$ then $\text{EXPTIME} = \Sigma_2^p$.

The previous theorem implies that if $P = \text{NP}$ then $\text{EXPTIME} \not\subseteq P/\text{poly}$. Thus upperbounds — albeit of a very strong kind — can be used to prove circuit lowerbounds.

6.2 Circuit lowerbounds

Theorem 6.1 implies that if $NP \not\subseteq P/\text{poly}$, then $P \neq NP$. The Karp-Lipton theorem gives hope that $NP \not\subseteq P/\text{poly}$. Can we resolve $P$ versus $NP$ by proving $NP \not\subseteq P/\text{poly}$? There is reason to invest hope in this approach as opposed to proving direct lowerbounds on Turing machines. By representing computation using circuits (Theorem 6.1) we seem to actually peer into the guts of it rather than treating it as a blackbox. Thus we may be able to get around the relativization results of Chapter 4.

Sadly, such hopes have not come to pass. After two decades, the best circuit size lowerbound for an $NP$ language is only $4n$. (However, see the Problems for a better lowerbound for a language in $\text{PH}$.) On the positive side, we have had notable success in proving lowerbounds for more restricted circuit models, as we will see in Chapter 14.

By the way, it is easy to show that for large enough $n$, almost every boolean function on $n$ variables requires large circuits.

Theorem 6.7
For $n \geq 100$, almost all boolean functions on $n$ variables require circuits of size at least $2^{n}/10n$.

Proof: We use a simple counting argument. There are at most $s^{3s}$ circuits of size $s$ (just count the number of labelled directed graphs, where each node has indegree at most 2). Hence this is an upperbound on the number of functions on $n$ variables with circuits of size $s$. For $s = 2^{n}/10n$, this number is at most $2^{2^{n}/10}$, which is miniscule compared to $2^{2^{n}}$, the number of boolean functions on $n$ variables. Hence most boolean functions do not have such small circuits. ∎

The problem with the above counting argument is of course, that it does not yield an explicit boolean function (say an $NP$ language) that requires large circuits.
6.3 Finer gradations among circuit classes

There are two reasons why subclasses of $P/poly$ are interesting. First, proving lowerbounds for these subclasses may give insight into how to separate $NP$ from $P/poly$. Second, these subclasses correspond to interesting computational models in their own right.

The most interesting such connection is to the massively parallel computers. In such a computer—a popular commercial product in the 1980s and 1990s—one uses simple off-the-shelf microprocessors and links them using an interconnection network that allows them to send messages to each other. Usual interconnection networks such as the hypercube allows linking $n$ processors such that interprocessor communication is possible—assuming some upperbounds on the total load on the network—in $O(\log n)$ steps. The processors compute in lock-step (for instance, to the ticks of a global clock) and are assumed to do a small amount of computation in each step, say an operation on $O(\log n)$ bits. Thus each processor computers has enough memory to remember its own address in the interconnection network and to write down the address of any other processor, and thus send messages to it. We are purposely omitting many details of the model (Leighton [Lei92] is the standard reference for this topic) since the validity of Theorem 6.8 below does not depend upon them.

6.3.1 Parallel computation and NC

**Definition 6.3** A computational task is said to have efficient parallel algorithms if inputs of size $n$ can be solved using a parallel computer with $n^{O(1)}$ processors and in time $\log^{O(1)} n$.

**Example 6.3** Given two $n$ bit numbers $x, y$ we wish to compute $x + y$ fast in parallel. The gradeschool algorithm proceeds from the least significant bit and maintains a carry bit. The most significant bit is computed only after $n$ steps. This algorithm does not take advantage of parallelism. A better algorithm called carry lookahead assigns each bit position to a separate processor and then uses interprocessor communication to propagate carry bits. It takes $O(n)$ processors and $O(\log n)$ time.

There are also efficient parallel algorithms for integer multiplication and division (the latter is quite nonintuitive and unlike the gradeschool algorithm!).
6.3. FINER GRADATIONS AMONG CIRCUIT CLASSES

EXAMPLE 6.4 Many matrix computations can be done efficiently in parallel; these include computing the product, rank, determinant, inverse, etc. (See exercises.)

Some graph theoretic algorithms such as shortest paths and minimum spanning tree also have fast parallel implementations.

But many well-known polynomial-time problems such as minimum matching, maximum flows, and linear programming are not known to have any good parallel implementations and are conjectured not to have any; see our discussion of P-completeness below.

Now we relate parallel computation to circuits. The depth of a circuit is the length of the longest directed path from an input node to the output node.

**Definition 6.4 (Nick’s class or NC)** A language is in NC$^i$ if there are constants $c, d > 0$ such that it can be decided by a logspace-uniform family of circuits $\{C_n\}$ where $C_n$ has size $O(n^c)$ and depth $O(\log^d n)$. The class NC is $\bigcup_{i \geq 1} NC^i$.

A related class is the following.

**Definition 6.5 (AC)** The class AC$^i$ is defined similarly to NC$^i$ except gates are allowed to have unbounded fanin. The class AC is $\bigcup_{i \geq 0} AC^i$.

Since unbounded (but poly($n$)) fanin can be simulated using a tree of ORs/ANDs of depth $O(\log n)$, we have NC$^i$ $\subseteq$ AC$^i$ $\subseteq$ NC$^{i+1}$, and the inclusion is known to be strict for $i = 0$ as we will see in later Chapter. (Notice, NC$^0$ is a trivial class since the circuit’s output depends upon a constant number of input bits, but AC$^0$ does not suffer from this limitation.)

**Example 6.5** The language PARITY =\{x : x has an odd number of 1s\} is in NC$^1$. The circuit computing it has the form of a binary tree. The answer appears at the root; the left subtree computes the parity of the first $|x|/2$ bits and the right subtree computes the parity of the remaining bits.
The gate at the top computes the parity of these two bits. Clearly, unwrapping the recursion implicit in our description gives a circuit of depth $O(\log n)$.

---

The classes AC, NC are important because of the following.

**Theorem 6.8**

A language has efficient parallel algorithms iff it is in NC.

**Proof:** Suppose a language $L \in NC$ and is decidable by a circuit family $\{C_n\}$ where $C_n$ has size $N = O(n^c)$ and depth $D = O(\log^d n)$. Take a general purpose parallel computer with $N$ nodes and configure it to decide $L$ as follows. Compute a description of $C_n$ and allocate the role of each circuit node to a distinct processor. (This is done once, and then the computer is ready to compute on any input of length $n$.) Each processor, after computing the output at its assigned node, sends the resulting bit to every other circuit node that needs it. Assuming the interconnection network delivers all messages in $O(\log N)$ time, the total running time is $O(\log^{d+1} N)$.

The reverse direction is similar, with the circuit having $N \cdot D$ nodes arranged in $D$ layers, and the $i$th node in the $t$th layer performs the computation of processor $i$ at time $t$. The role of the interconnection network is played by the circuit wires. $\square$

A major open question in this area is whether $P = NC$. We believe that the answer is No (though we are currently even unable to separate PH from NC$^1$). This motivates the theory of $P$-completeness, a study of which problems are likely to be in NC and which are not.

### 6.3.2 $P$-completeness

**Definition 6.6** A language is $P$-complete if it is in $P$ and every language in $P$ is logspace-reducible to it.

The following easy theorem is left for the reader.

**Theorem 6.9**

If language $L$ is $P$-complete then $L \in NC$ (resp., $L$) iff $P = NC$ (resp., $P = L$).

Consider the following language.

$\text{CIRCUIT-VALUE} = \{<C, x>: C$ is a circuit and $x$ is an input such that $C(x) = 1\}$. \hspace{1cm}(6.5)$
6.4. CIRCUITS OF EXPONENTIAL SIZE

**Theorem 6.10**

*CIRCUIT-VALUE* is \( P \)-complete.

**Proof:** The language is clearly in \( P \). A logspace-reduction from any other language in \( P \) to this language is implicit in the proof of Theorem 6.2. \( \square \)

### 6.4 Circuits of exponential size

As noted, every language has circuits of size \( O(n2^n) \). However, computing these circuits may difficult—and impossible if the language is undecidable.

If we place a uniformity condition on the circuits, that is, require them to be efficiently computable then the circuit complexity of some languages could exceed \( n2^n \). In fact it is possible to give alternative definitions of some familiar complexity classes, analogous to the definition of \( P \) in Theorem 6.2.

**Definition 6.7 (DC-Uniform)** Let \( \{C_n\}_{n \geq 1} \) be a circuit family of size \( S(n) \geq n \) for input of length \( n \). We say that this is a Direct Connect uniform (DC uniform) family if all the following functions can be computed in deterministic \( O(\text{poly}(n)\log(S(n))) \) time:

- \( \text{TYPE}(n, i) = \) the type (AND, OR, NOT, INPUT, OUTPUT) of gate \( i \) in circuit \( C_n \).
- \( \text{IN}(n, i, j) = k \) where \( k \) is \( j^{th} \) in the ordered list of indices to the gates that feed into gate \( i \), \( k = \text{NONE} \) if there is no \( j^{th} \) gate.
- \( \text{OUT}(n, i, j) = k \) where \( k \) is \( j^{th} \) in the ordered list of indices to the gates that gate \( i \) feeds into, \( k = \text{NONE} \) if there is no \( j^{th} \) gate.

Note that the circuits may have exponential size, but they have a succinct representation in terms of a TM which can systematically generate any required node of the circuit in polynomial time.

Now we give a (yet another) characterization of the class \( \text{PH} \), this time as languages computable by uniform circuit families of bounded depth. We leave it as an exercise.

**Theorem 6.11**

\( L \in \text{PH} \) iff \( L \) can be computed by a DC uniform circuit family \( \{C_n\} \) that

- uses AND, OR, NOT gates.
- has size \( 2^{n^{O(1)}} \) and constant depth (i.e., depth \( O(1) \)).
• gates can have unbounded (exponential) fanin.

• the NOT gates appear only at the input level.

If we drop the restriction that the circuits have constant depth, then we obtain exactly EXPTIME (see Exercises).

Exercises

Kannan [Kan82] Show for every $k > 0$ that PH contains languages whose circuit complexity is $\Omega(n^k)$. (Hint: Keep in mind the proof of the existence of functions with high circuit complexity.)

§1 Solve the previous question with PH replaced by $\Sigma^p_2$.

§2 (KL82, attributed to A. Meyer) Show that if $\text{EXPTIME} \subseteq \text{P/poly}$ then $\text{EXPTIME} = \Sigma^p_2$.

§3 Show that if $\text{P} = \text{NP}$ then there is a language in $\text{EXPTIME}$ that requires circuits of size $2^n/n$.

§4 Show that if some unary language is NP-complete, then $\text{P} = \text{NP}$.

§5 (Open) Suppose we make a stronger assumption than $\text{NP} \subseteq \text{P/poly}$: every language in $\text{NP}$ has linear size circuits. Can we show something stronger than $\text{PH} = \Sigma^p_2$?

§6 (a) Describe an NC circuit for the problem of computing the product of two given $n \times n$ matrices $A, B$.

(b) Describe an NC circuit for computing, given an $n \times n$ matrix, the matrix $A^n$. (Hint: Use repeated squaring: $A^{2^k} = (A^{2^{k-1}})^2$.)

(c) Conclude that the PATH problem (and hence every NL language) is in NC. (Hint: What is the meaning of the $(i, j)$th entry of $A^n$?)

§7 A formula is a circuit in which every node (except the input nodes) has outdegree 1. Show that a language is computable by polynomial-size formulae iff it is in (nonuniform) $\text{NC}^1$. (Hint: a formula may be viewed —once we exclude the input nodes—as a directed binary tree, and in a binary tree of size $m$ there is always a node whose removal leaves subtrees of size at most $2m/3$ each.)
§8 Show that $NC^1 = L$. Conclude that $PSPACE \neq NC^1$.

§9 Show that $\text{EXPTIME}$ is exactly the set of languages with $\text{DC}$ uniform circuits of size $2^{nc}$ where $c$ is some constant ($c$ may depend upon the language).

§10 Show that if linear programming has a fast parallel algorithm then $P = NC$. (Hint: in your reduction, express the $\text{CIRCUIT-VALUE}$ problem as a linear program and use the fact that $x \lor y = 1$ iff $x + y \geq 1$. Be careful; the variables in a linear program are real-valued and not boolean!)

Chapter notes

Karp-Lipton theorem is from [KL82]. The class of $\text{NC}$ algorithms as well as many related issues in parallel computation are discussed in Leighton [?].
Chapter 7

Randomized Computation

For some computational problems, allowing the algorithm to make probabilistic choices seems to greatly speed up the computation. This chapter studies the power of probabilistic Turing Machines (PTMs), which are Turing machines provided with the ability to make probabilistic choices. We will assume that such a probabilistic choices are statistically random, ignoring philosophical issues having to do with whether or not such randomness exists in the world. (Later in Chapter 18 we return to the issue of imperfect sources of randomness.)

A probabilistic Turing Machine looks no different from a nondeterministic TM: up to two transitions are possible out of each state. Roughly speaking, the difference lies in how we interpret the tree of all possible computations: instead of asking whether the TM enters the “accept” state $q_{\text{acc}}$ on some path, we ask how large is the fraction of paths on which this happens.

![State transition diagram of a probabilistic TM]

Figure 7.1: The state transition diagram of a probabilistic TM looks just like that of an NDTM (see Figure 1.2) but the semantics differ.

More precisely, we assume that if for any state $q$ and symbol $b$, there are two possible outgoing transitions for the machine, then we think of $q$ as a
“coin-tossing” state. We assume the machine randomly (and equiprobably) chooses one of the two transitions. This random choice is independent of all other random choices made in the past. In terms of the tree of all configurations, this means that whenever the machine reaches a branchpoint, it can go either way with probability \(1/2\). Thus if some configuration occurs \(k\) levels down from the root (i.e., the start configuration), the probability the machine arrives in that configuration is \(1/2^k\). Another view of this process is that the PTM has the ability to \textit{randomly choose} between \(2^k\) different options in \(O(k)\) time.

![Figure 7.2: A probabilistic TM has \(2^k\) possible configurations after \(k\) steps, and they are equally likely.](image)

The quantity of interest is

\[
\sum_{\text{leaf nodes where machine is in } q_{\text{acc}}} \Pr[\text{Machine arrives at this leaf node}].
\]

This quantity is the probability that the machine accepts the input.

Notice that our everyday model of a randomized algorithm, a C program with a (perfect) random number generator, is equivalent to the above model since a random number generator that produces random \(k\)-bit integers can be simulated on a PTM by using random choices to write down a random string of \(k\) bits. Conversely, a C program with a random number generator can easily simulate a PTM.

In fact, assuming true random sources exist, a polynomial-time PTM is arguably a more realistic realization of what we might mean by “feasible” computation than a polynomial-time deterministic TM. Thus the classes \(\text{RP, BPP}\) introduced in this chapter are almost as important as \(\text{P}\). However, we note that the role of randomness in complexity theory extends far beyond a study of randomized algorithms or \(\text{RP}\) and \(\text{BPP}\). Entire areas such as cryptography, interactive and probabilistically checkable proofs, etc. rely on randomness in an essential way, sometimes to prove results whose
statement did not call for randomness at all. The groundwork for studying those areas will be laid in this chapter.

Finally, we should mention that randomness has revolutionized algorithm design in the past two decades, leading to exact and approximate algorithms in a host of fields—from combinatorial algorithms to algebraic computation to network routing. In some cases, we know how to “derandomize” the algorithm to obtain a deterministic algorithm, but that doesn’t change the fact that the main idea—and often the simplest implementation—is still probabilistic.

7.1 RP, coRP, BPP, ZPP

We say that a randomized machine runs in time $t(n)$ if for every input of length $n$, every branch in the computation tree terminates in $O(t(n))$ steps. Now we define analogues of $P$ for probabilistic computation.

**Definition 7.1** $RTIME(t(n))$ contains every language $L$ for which there is a probabilistic TM $M$ running in $t(n)$ time such that

$$x \in L \Leftrightarrow \Pr[M \text{ accepts } x] \geq \frac{1}{2}$$  \hspace{1cm} (7.1)

$$x \notin L \Leftrightarrow \Pr[M \text{ accepts } x] = 0$$  \hspace{1cm} (7.2)

$BPTIME(t(n))$ is defined similarly, except

$$x \in L \Leftrightarrow \Pr[M \text{ accepts } x] \geq \frac{2}{3}$$  \hspace{1cm} (7.3)

$$x \notin L \Leftrightarrow \Pr[M \text{ accepts } x] \leq \frac{1}{3}$$  \hspace{1cm} (7.4)

We define $RP = \bigcup_{c \geq 0} RTIME(n^c)$ and $BPP = \bigcup_{c \geq 0} BPTIME(n^c)$. Finally, $coRP = \{\overline{L} : L \in RP\}$.

It is helpful to note that the RP machines exhibit one-sided error. If we run the machine and it ends up in an accepting state, then we are sure that $x \in L$. But if the machine is observed to halt without accepting, we cannot be sure that $x \notin L$. By the same token, BPP machines exhibit two-sided error. A corollary to the above observation is that $RP \subseteq NP$, since every accepting branch is a “certificate” that the input is in the language. This reasoning does not apply to BPP machines because of 2-sided error. We do not know if $BPP \subseteq NP$, though Theorem 7.6 will show $BPP \subseteq PH$. Notice, BPP is closed under complementation (if $L \in BPP$ then $\overline{L} \in BPP$) which doesn’t appear to be true of NP. (Furthermore,
BPTIME\( (t(n)) \subseteq \text{DTIME}(2^{O(t(n))}) \) since the entire tree of configurations can be deterministically searched in \( 2^{O(t(n))} \) time.

Now we define PTMs that never err (also called “zero error” machines).

**Definition 7.2** A PTM runs in expected time \( t(n) \) if for every input of size \( n \), the following expectation is \( O(t(n)) \):

\[
\sum_{\text{leaf node } b} \Pr[\text{machine ends up in } b] \times (\text{number of steps taken to reach } b).
\]

\( \text{ZTIME}(t(n)) \) contains every language \( L \) for which there is a machine \( M \) that runs in expected time \( t(n) \) such that the machine never accepts (i.e., never goes into \( q_{\text{acc}} \)) on inputs \( x \notin L \) and never rejects (i.e., never goes into \( q_{\text{halt}} \)) on inputs \( x \in L \). We define \( \text{ZPP} = \cup_{c>0} \text{ZTIME}(nc) \).

The next theorem is left as exercise. It ought to be slightly surprising, since the corresponding statement for nondeterminism is open; i.e., whether or not \( P = \text{NP} \cap \text{coNP} \).

**Theorem 7.1**

\( \text{ZPP} = \text{RP} \cap \text{coRP} \).

### 7.1.1 The robustness of our definitions

When we defined \( P \) and \( \text{NP} \), we argued that our definitions are robust and were likely to be the same for an alien studying the same concepts in a faraway galaxy. Now we address similar issues for probabilistic computation.

**The role of the precise constants 1/3, 1/2 etc.** Changing these constants has no effect.

**Claim 7.2**

If in the definition of \( \text{BPP} \) we were to substitute \( \rho + 1/n^c \) and \( \rho - 1/n^c \) for \( 2/3 \) and \( 1/3 \) respectively (where \( \rho, c \) are any constants), then the class would be unchanged.

**Proof:** If \( L \) is decided by a PTM \( M \) under this new definition, then consider a PTM \( M' \) that given any input runs \( M \) about \( N = n^{2c+1} \) times (using new random bits every time) and accepts iff \( M \) accepts more than \( N \cdot (\rho+0.01/n^c) \) times. By Chernoff bounds (Theorem A.10 in the appendix), if \( x \in L \) then \( M \) accepts with probability \( 1 - \exp(-n) \) and if \( x \notin L \) then \( M \) accepts with probability at most \( \exp(-n) \). Thus \( M' \) still runs in polynomial time but decides \( L \) under the old definition. \( \square \)
Expected running time versus worst-case running time. When defining $\text{RTIME}(t(n))$ and $\text{BPTIME}(t(n))$ we required every branch to halt in $t(n)$ time. We could have used expected running time instead, as we later did for Definition 7.2. This also would not change the classes RP, BPP since we could add a time counter to the machine and make it halt after $100t(n)$ steps. By simple averaging, the probability is at most $1/100$ that the machine enters a branch that runs for more than $100t(n)$ steps. Thus the acceptance probability is changed by at most $1/100$, which by Claim 7.2 does not affect the class.

Allowing more general random choices than a fair random coin.
One could conceive of real-life computers that have a “coin” that comes up heads with probability $\rho$ that is not $1/2$. In fact it is conceivable that for a random source based upon quantum mechanics, $\rho$ is an irrational number, such as $1/e$. Could such a coin give probabilistic algorithms new power? The following claim shows that it will not.

Claim 7.3
A coin with $\Pr[\text{Heads}] = \rho$ can be simulated by a PTM in expected time $O(1)$ provided the $i$th bit of $\rho$ is computable in $\text{poly}(i)$ time.

Proof: Let the binary expansion of $\rho$ be $0.p_1p_2p_3\ldots$. The PTM generates a sequence of random bits $b_1, b_2, \ldots$, one by one, where $b_i$ is generated at step $i$. If $b_i < p_i$ then the machine outputs “heads” and stops; if $b_i > p_i$ the machine outputs “tails” and halts; otherwise the machine goes to step $i + 1$. Clearly, the machine reaches step $i + 1$ iff $b_j = p_j$ for all $j \leq i$, which happens with probability $1/2^i$. Thus the probability of “heads” is $\sum_i p_i \frac{1}{2^i}$, which is exactly $\rho$. Furthermore, the expected running time is $\sum_i i^c \cdot \frac{1}{2^i}$. For every constant $c$ this infinite sum is upperbounded by another constant. (See exercises.) $\square$

7.1.2 Alternative definitions
As in the case of NP, we can given an alternative definiton for these classes. Instead of the TM tossing coins by itself, we can think of the sequence of coin flips as an additional input to the TM. Then the TM can be deterministic. Notice, if the machine runs for polynomial time, then the string of coin tosses has polynomial size too.
Definition 7.3 (BPP, alternative definition) BPP contains a language $L$ iff there is a language $L_0 \in P$ and a constant $c > 0$ such that

$$x \in L \Leftrightarrow \Pr_{r \in \{0,1\}^{|x|^c}} [(x, r) \in L_0] \geq \frac{2}{3} \quad (7.5)$$

$$x \notin L \Leftrightarrow \Pr_{r \in \{0,1\}^{|x|^c}} [(x, r) \in L_0] \leq \frac{1}{3} \quad (7.6)$$

RP can be similarly redefined.

### 7.1.3 Error reduction and recycling of random bits

As noted above, the constants $2/3$ and $1/3$ in the definition of BPP play no special role. Claim 7.2 shows that these can be replaced by any pair of numbers where the first is at least $1/n^c$ more than the other, for some fixed constant $c$. Furthermore, the gap of $1/n^c$ in the two cases can be “boosted” so that they become $1 - \exp(-\text{poly}(n))$ and $\exp(-\text{poly}(n))$ respectively. For all practical purposes, such probabilities can be thought of as 1 and 0 respectively. (In other words, for all practical purposes, there is little difference between BPP and ZPP, even though they seem to very distinct to a complexity theorist.)

The transformation described in Claim 7.2 is called error reduction. It involves running the PTM for $\text{poly}(n)$ independent runs using independent random bits each time. Ideally, one would like to be frugal with using randomness, because good quality random number generators tend to be slower than the rest of the computer. Surprisingly, the error reduction can be done just as effectively without using truly independent runs, and “recycling” the random bits. Now we outline this idea; a much more general theory will be later presented in Chapter 18.

We explain the idea with respect to an RP machine $M$ but a similar idea applies to BPP machines too. If we used a random string $r$ to run $M$ once and the machine failed to detect $x$ being in $L$, then all we know about the random string used is that it is in the set of bad random strings. Thus it still has a lot of “randomness” left in it. The idea in “recycling” is to reused this randomness. One method is to use expander graphs.

Definition 7.4 If $d > 2$ and $\beta > 0$, a family of $(d, \beta)$-expanders is a sequence of graphs $\{G_n\}$ where $G_n$ is a $d$-regular graph on $i$ nodes, and has the property that every subset $S$ of at most $n/2$ nodes has edges to at least $\beta |S|$ nodes outside $S$.

Deep mathematics (and more recently, simpler mathematics) has been used to construct expanders. These constructions yield algorithms that,
given \( n \) and an index of a node in \( G_n \), can produce the indices of the neighbors of this node in \( \text{poly}(\log n) \) time.

Suppose the RP algorithm uses \( m \) random bits. The naïve approach to reduce its error probability to \( 2^{-k} \) uses \( O(mk) \) random bits. A better approach involving recycling is as follows. Pick the first random string as a node in an expander graph on \( 2^m \) nodes, take a random walk of length \( k \) from there, and use the indices of the nodes encountered in this walk (a total of \( mk \) bits) in your RP algorithm. Since the graph has constant degree, each step of the random walk needs \( O(1) \) random bits. So a random walk of length \( k \) takes \( m + O(k) \) random bits; \( m \) to produce the first node and \( O(1) \) for each subsequent step. A surprising result shows that this sequence will be random enough to guarantee the probability amplification, but we will postpone a proof to Section 18.3.3.

### 7.2 Some examples of PTMs

**Example 7.1** Let \( \text{COMPOSITES} \) be the language \( \{ [n]_p : n \) is a composite integer \( \} \). Thus \( \text{COMPOSITES} \) is \( \text{PRIMES} \), the set of encodings of prime numbers.

Designing tests for primality is an old hobby of mathematicians and algorithms designers. Here we give a simple test showing \( \text{COMPOSITES} \) is in RP.

We recall a well-known fact. Say \( a \) is a residue mod \( n \) iff there is a \( b \in \mathbb{Z}_n \) such that \( b \equiv a^2 \mod n \). It is well-known that if \( p \) is a prime, then

\[
a_p^{\frac{p-1}{2}} \equiv \begin{cases} 1 \mod p & \text{if } a \text{ is a residue mod } p \\ -1 \mod p & \text{if } a \text{ is not a residue mod } p \end{cases}
\]

(7.7)

One would like to use this fact as a basis for recognizing primes, say by testing the above condition for random \( a \in \mathbb{Z}_p^\ast \). Notice that the input size is \( \log p \) (or \( \log n \)) and we need to compute \( a_p^{\frac{p-1}{2}} \) in polynomial time. This can be done using fast exponentiation; see exercises.

However, the test we just hinted at can still get fooled by some composite numbers. A better test uses the so-called Jacobi symbol that maps pairs of positive integers \( p, q \) to \( \{-1, 1\} \). The value of the function on \( p, q \) is denoted \( \left( \frac{a}{b} \right) \). This function is computable in polynomial time, and when \( q \) is prime it is 1 if \( p \) is a residue and -1 if \( p \) is not a residue.

Here is the better test: “Pick a random number \( a \in \mathbb{Z}_n \). Compute \( \gcd(a, n) \) and if it is not 1 (i.e., \( a, n \) have a common divisor) then output “Accept.”
Else compute $a^{(n-1)/2} \pmod{n}$ using fast exponentiation and also $(\frac{a}{n})$. If the two are different, output “Accept” else output “Reject.”

The reader may wish to consult a book in elementary number theory to understand the properties of the Jacobi symbol and then doublecheck that the above algorithm works.

We remark that when the algorithm outputs “Accept,” we know that $n$ is composite but obtain no information about the factors of $n$! Indeed, finding factors is the famous FACTORING problem, which is conjectured to have no polynomial-time algorithms whose (conjectured) difficulty underlies famous cryptosystems. (See Chapter 9.)

---

**Example 7.2** In this example, a “polynomial” refers to a multivariate polynomial with integer coefficients whose representation can use $+, \times, -$ and parentheses, e.g. $(1 + x_1)(1 - x_2 x_3^5 + x_1) - x_3^5 + x_4(1 + x_2)(1 - x_3)$. Checking whether such a polynomial is identically zero seems computationally difficult: we may need to open all parentheses, which may produce exponentially many terms. Surprisingly, a simple randomized algorithm exists. We claim that the language

$$NONZEROPOLYNOMIAL = \{p : p \text{ is a nonzero polynomial}\}$$

is in $\text{RP}$.

Suppose some input for this problem has $n$ bits. Then it is easy to see that the total degree $d$ of the polynomial is at most $2^n$. (Total degree is the maximum degree of any monomial; e.g., the degree of $x_1^3 x_2^4$ is 7.) The algorithm is very simple. For each variable substitute a random integer value picked from $[1, 2d]$, substitute all of these into the polynomial, and simplify. It is easy to check that the evaluation runs in $\text{poly}(n)$ time. If the polynomial evaluates to a nonzero value, the algorithm accepts and otherwise it outputs reject. The correctness of this algorithm follows from the next Lemma, whose proof appears in the appendix.

---

**Lemma 7.4 (Schwartz-Zippel)**

Let $p(x_1, x_2, \ldots, x_m)$ be a polynomial of total degree at most $d$ and $S$ is any finite set of integers. When $a_1, a_2, \ldots, a_m$ are randomly chosen with
replacement from $S$, then

$$\Pr[p(a_1, a_2, \ldots, a_m) \neq 0] \geq 1 - \frac{d}{|S|}.$$  

**Example 7.3** If $G = (V_1, V_2, E)$ is the bipartite graph where $|V_1| = |V_2|$ and $E \subseteq V_1 \times V_2$ then a perfect matching is some $E' \subseteq E$ such that every node appears exactly once among the edges of $E'$. Alternatively, we may think of it as a permutation $\sigma$ on the set $\{1, 2, \ldots, n\}$ (where $n = |V_1|$) such that for each $i \in \{1, 2, \ldots, n\}$, the pair $(i, \sigma(i))$ is an edge. Many deterministic algorithms are known for detecting if a perfect matching exists. Here we describe a very simple randomized algorithm (due to Lovász) using the Schwartz-Zippel lemma.

Consider the $n \times n$ matrix $X$ (where $n = |V_1| = |V_2|$) whose $(i, j)$ entry $X_{ij}$ is the variable $x_{ij}$ if $(i, j) \in E$ and 0 otherwise. Recall that the determinant of matrix $\det(X)$ is

$$\det(X) = \sum_{\sigma \in S_n} (-1)^{\text{sign}(\sigma)} \prod_{i=1}^{n} X_{i,\sigma(i)},$$

(7.8)

where $S_n$ is the set of all permutations of $\{1, 2, \ldots, n\}$. Note that every permutation is a potential perfect matching, and the corresponding monomial in $\det(X)$ is nonzero iff this perfect matching exists in $G$. Thus the graph has a perfect matching iff $\det(X) \neq 0$.

Now observe two things. First, the polynomial in (7.8) has $|E|$ variables and total degree at most $n$. Second, even though this polynomial may be of exponential size, for every setting of values to the $X_{ij}$ variables it can be efficiently evaluated, since computing the determinant of a matrix with integer entries is a simple polynomial-time computation (actually, even in NC$^2$).

This leads us to Lovász's randomized algorithm: pick random values for $X_{ij}$’s from $[1, \ldots, 2n]$, substitute them in $X$ and compute the determinant. If the determinant is nonzero, output “accept” else output “reject.” The advantage of the above algorithm over classical algorithms is that it can be implemented by a randomized NC circuit, which means (by the ideas of Section 6.3.1) that it has a fast implementation on parallel computers.
Example 7.4 In the Polynomial Identity testing problem we are given two polynomials that are computed by algebraic circuits (over the integers) of size \( n \), and have to decide if they are the same. Note that the circuits compute multivariate polynomial of degree at most \( 2^n \). We can pick a random number \( p \) of \( O(n) \) bits and random values of the inputs mod \( p \). Then evaluate the circuit mod \( p \). Repeat poly(\( n \)) times. needs more detail Thus the problem is in \( \text{RP} \).

Example 7.5 (Fingerprinting) This example is not really a language in \( \text{RP} \) or \( \text{BPP} \), but a general paradigm for designing randomized algorithms. Suppose two parties each hold an \( n \)-bit integer, \( x \) and \( y \) respectively. They wish to determine if \( x \neq y \). The tedious deterministic way to do this is for them to exchange all \( n \) bits of their inputs. (In Chapter 13 we will prove that no deterministic algorithm can do any better.)

Here is a randomized way: they randomly pick a prime \( p \) in the interval \([1, 3n \ln n]\). They compute \( x_1 = x \pmod{p} \) and \( y_1 = y \pmod{p} \) and exchange \( x_1, y_1 \). (Notice, \( p \) has at most \( 2 \log n \) bits, and hence so do \( x_1, y_1 \).) If \( x_1, y_1 \) are found to be different, the two parties can safely conclude that \( x, y \) are different.

We claim that if \( x \neq y \), then the probability is at least \( 1/2 \) that the two parties will detect this. Why?

First, the number \( x - y \) is at most \( 2^n \) in magnitude, so it has at most \( n \) distinct prime factors. Second, according to the Prime Number Theorem, the number of primes in the interval \([1, m]\) is asymptotically \( m/\ln m \). Thus the number of primes in \([1, 3n \ln n]\) is approaches \( 3n \) as \( n \) gets large enough, so concretely say it is at least \( 2n \). When \( p \) is picked randomly among these primes, the event \( x \equiv y \pmod{p} \) happens iff \( x - y \equiv 0 \pmod{p} \), in other words iff \( p \mid (x - y) \). The chance of this happening is at most \( n/2n = 1/2 \).

This technique is called fingerprinting because the \( n \)-bit integer is represented by a \( O(\log n) \)-bit residue, reminiscent of the way that a person’s fingerprint can be an essentially unique (though not foolproof) identifier for him/her.
7.3 BPP ⊆ P/poly

Now we show that all BPP languages have polynomial sized circuits. Together with Theorem 6.4 this implies that if 3SAT ∈ BPP then PH = Σ²

**Theorem 7.5**

BPP ⊆ P/poly.

**Proof:** Suppose L ∈ BPP, and M is a randomized TM for L that on inputs of size n uses m random bits and satisfies x ∈ L ⇒ Pr_r [M(x, r) accepts] ≥ 1 − 2^{−(n+1)} and x /∈ L ⇒ Pr_r [M(x, r) accepts] ≤ 2^{−(n+1)}. (Such an M exists by the error reduction arguments mentioned earlier.)

Say that an r ∈ {0, 1}^m is bad for x if M(x, r) is an incorrect answer. For each x, at most 2^m/2^{(n+1)} values of r are bad. Adding over all x, we conclude that at most 2^n × 2^m/2^{(n+1)} = 2^m/2 strings r are bad for some x. In other words, at least 2^m − 2^m/2 choices of r are not bad for every x. Let r_0 be any such choice for r. Then M(x, r_0) is the correct answer for every x. We can hard-wire this r_0 into a polynomial size circuit. Given an input x, the circuit simulates M on (x, r_0). Hence we have shown L ∈ P/poly. □

7.4 BPP is in PH

The next theorem relates BPP to the polynomial hierarchy.

**Theorem 7.6**

BPP ⊆ Σ² ∩ Π²

**Proof:** It is enough to prove that BPP ⊆ Σ² because BPP is closed under complementation.

Suppose L ∈ BPP, and M is a randomized TM for L that uses m random bits and satisfies x ∈ L ⇒ Pr_r [M(x, r) accepts] ≥ 1 − 2^{-n} and x /∈ L ⇒ Pr_r [M(x, r) accepts] ≤ 2^{-n}.

Fix an input x, and let S_x denote the set of r for which M accepts (x, r). Then either |S_x| ≥ (1 − 2^{-n})2^m or |S_x| ≤ 2^{-n}2^m, depending on whether or not x ∈ L. We will show how to check, using two alternations, which of the two cases is true.

Consider r as an element of GF(2)^m. For a set S ⊆ GF(2)^m we define the shift of S by r_0 as \{r + r_0 | r ∈ S\} (where the addition is as in GF(2)^m, i.e., bit-wise XOR).

Suppose x ∈ L, so |S_x| ≥ (1 − 2^{-n})2^m. We shall show that GF(2)^m can be covered using a small number of shifts of S_x.
Figure 7.3: There are only two possible sizes for the set of \( r \) such that \( M(x, r) = \text{Accept} \): either this set is almost all of \( \{0, 1\}^m \) or a tiny fraction of \( \{0, 1\}^m \). In the former case, a few random “shifts” of this set are quite likely to cover all of \( \{0, 1\}^m \).

**Lemma 7.7**
If \( |S_x| \geq (1 - 2^{-n})2^m \) then there are \( r_1, r_2, \ldots, r_k \in \text{GF}(2)^m \), where \( k = \lceil \frac{m}{n} \rceil + 1 \) such that \( \bigcup_{i=1}^{k} (S_x + r_i) = \text{GF}(2)^m \).

**Proof:** We use the probabilistic method: pick \( r_1, r_2, \ldots, r_k \) randomly and show that \( \Pr_{r_1, r_2, \ldots, r_k} \left[ \bigcup_{i=1}^{k} (S_x + r_i) \neq \text{GF}(2)^m \right] < 1/2 \). Then the existence of the desired \( r_1, r_2, \ldots, r_k \) has been proved.

Let \( z \in \text{GF}(2)^m \). If \( r_1 \) is uniformly distributed among all vectors in \( \text{GF}(2)^m \), then so is \( z + r_1 \). So \( \Pr_{r_1} [z \notin S_x + r_1] = \Pr_{r_1} [z + r_1 \notin S_x] \leq 2^{-n} \).

Since \( r_1, r_2, \ldots, r_k \) were chosen independently of one another, we have for each \( z \),

\[
\Pr_{r_1, r_2, \ldots, r_k} \left[ z \notin \bigcup_{i=1}^{k} S_x + r_i \right] \leq 2^{-nk}
\]

So, \( \Pr_{r_1, r_2, \ldots, r_k} \left[ \text{some } z \notin \bigcup_{i=1}^{k} S_x + r_i \right] \leq 2^{m-nk} < 1/2 < 1 \).

Now suppose \( x \notin L \). Then \( |S_x| \leq 2^{-n}2^m \), which is a tiny set. The union of any \( k \) shifts of \( S_x \) can have size at most \( k2^{m-n} < 2^m \), and hence cannot equal \( \text{GF}(2)^m \).

Thus we have established that

\[
x \in L \iff \exists r_1, r_2, \ldots, r_k \in \text{GF}(2)^m \text{ such that } \forall z \in \text{GF}(2)^m \quad M(x, y) = \text{"accept" for some } y \in \{z + r_1, z + r_2, \ldots, z + r_k\} \quad (7.9)
\]

Thus \( L \) is in \( \Sigma_2^p \). \( \square \)
7.5 State of our knowledge about BPP

We know that $P \subseteq \text{BPP} \subseteq P/poly$, and furthermore, that $\text{BPP} \subseteq \Sigma_2^P \cap \Pi_2^P$. There are complexity-theoretic reasons to strongly believe (see our discussion of derandomization before Theorem 9.8) that $\text{BPP} \subseteq \text{DTIME}(2^{\epsilon n})$ for every $\epsilon > 0$, and in fact to reasonably suspect that in fact $\text{BPP} = P$ (see Chapter 18). However, currently we are even unable to rule out that $\text{BPP} \neq \text{NEXPTIME}$.

7.5.1 Complete problems for BPP?

No language is known to be complete for BPP under deterministic polynomial time reductions. It is conjectured that none exist. The reason is that the defining property of BPTIME machine is semantic, namely, that for every string they either accept with probability at least $2/3$ or reject with probability at least $1/3$. Given the description of a Turing machine $M$, testing whether it has this property is undecidable. By contrast, the defining property of an NDTM is syntactic: its finite control features nondeterminism. Completeness seems easier to define for syntactically defined classes than for semantically defined ones. (One can define a complete problem for BPP involving promise problems instead of languages, but we will not explore that.)

7.5.2 Does BPTIME have a hierarchy theorem?

Is $\text{BPTIME}(n^c)$ contained in $\text{BPTIME}(n)$? One would imagine not, but currently we are even unable to show that $\text{BPTIME}(2^{\sqrt{n}})$ (say) is not in $\text{BPTIME}(n)$. The standard diagonalization techniques fail, mostly because we do not know if $\text{BPTIME}(t(n))$ has a complete problem.

**Definition 7.5 (Strongly hard)** A language $L$ is strongly hard for $\text{BPTIME}(t(n))$ where $t(n) > n$ if there is a $c > 0$ such that every language $L' \in \text{BPTIME}(t(n))$ is reducible to $L$ via a reduction that runs in time $O(t(n)^c)$.

**Example 7.6** One can similarly define strongly hard languages for $\text{NTIME}(t(n))$. The standard proof of the Cook-Levin theorem shows that 3SAT is a strongly hard $\text{NTIME}(t(n))$. 
Lemma 7.8
If $\text{BPTIME}(n)$ has a language that is strongly hard for it then there is a constant $c > 0$ such that $\text{BPTIME}(t(n)) \not\subseteq \text{BPTIME}(t(n)^c)$.

Proof: (sketch) to be written $\square$

Recently Barak made some progress towards a hierarchy theorem; see the notes at the end of the chapter.

7.6 Randomized reductions

Since we have defined randomized algorithms, it also makes sense to define a notion of randomized reduction between two languages. This proves useful in some complexity settings, e.g., in the next two chapters.

Definition 7.6 Language $A$ reduces to language $B$ under a randomized polynomial time reduction, denoted $A \leq_r B$, if there exists a deterministic, polynomial time computable function $f : \{0,1\}^* \rightarrow \{0,1\}^*$ and a polynomial $p()$ such that

$$\forall x \in A \text{ Pr}_{y \in \{0,1\}^{p(|x|)}}[f(x, y) \in B] \geq \frac{2}{3} \quad (7.10)$$
$$\forall x \notin A \text{ Pr}_{y \in \{0,1\}^{p(|x|)}}[f(x, y) \in B] \leq \frac{1}{3} \quad (7.11)$$

We note that if $A \leq_r B$ and $B \in \text{BPP}$ then $A \in \text{BPP}$. This alerts us to the possibility that we could have defined NP-completeness using randomized reductions instead of deterministic reductions. Doing this would still have resulted in an interesting theory, since many people would consider a randomized polynomial-time algorithm for SAT to be just as “practical” as a deterministic one. However, Theorem 7.5 and Theorem 6.4 shows that this is an unlikely scenario: if 3SAT $\in$ BPP then SAT $\in \text{P/poly}$, implying $\text{PH} = \Sigma_2^P$.

Nevertheless, it is interesting to study randomized reductions. Recall that the Cook-Levin theorem shows that NP may be defined as the set \{ $L : L \leq_p 3\text{SAT}$ \}. The following definition is analogous.

Definition 7.7 $(\text{BP} \cdot \text{NP})$ $\text{BP} \cdot \text{NP} = \{ L : L \leq_r 3\text{SAT} \}$.

We explore the properties of $\text{BP} \cdot \text{NP}$ in the exercises, including whether or not $3\text{SAT} \in \text{BP} \cdot \text{NP}$.
7.7 Randomized space-bounded computation

A PTM is said to work in space \( S(n) \) if every branch requires space \( O(S(n)) \) on inputs of size \( n \) and terminates in \( 2^{O(S(n))} \) time\(^1\). Recall that the machine has a read-only input tape, and a work tape. Since it is a PTM, the transition diagram has some states with two outgoing transitions, which can be taken with equal probability. The most interesting case is when the worktape has \( O(\log n) \) size and the associated complexity classes \( RL \) and \( BPL \) are defined analogously to \( RP \) and \( BPP \).

As in the case of \( RP \) and \( BPP \), we can give an alternative definition for \( RL \), \( BPL \) by thinking of the sequence of random coin tosses as an additional input. However, some care is needed. A probabilistic logspace machine may use \( \text{poly}(n) \) bits during its computation, but it does not have space to “remember” all these bits. Thus the proper way to think of the random coin tosses is as an additional input to which the machine has one way access: it can read this string from left to right one bit at a time, but it cannot go right to left to reread any bits. (Notice, this viewpoint drives home the point that the machine’s workspace is too small for it to have much “memory” of the random string seen so far. This suggests that we can recycle the string in some way, this is indeed true, as we will see in Section 18.4.4.

**Definition 7.8** \( BPL \) is the class of all languages \( L \), such that there is a polynomial time randomized TM \( M \) with \( O(\log n) \) space and one-way access to a random string \( r \) such that

\[
\begin{align*}
\Pr_{r \in \{0,1\}^{|x|}} [M \text{ accepts } (x, r)] & \geq \frac{2}{3} \quad (7.12) \\
1 - \Pr_{r \in \{0,1\}^{|x|}} [M \text{ accepts } (x, r)] & \leq \frac{1}{3} \quad (7.13)
\end{align*}
\]

We note that \( RL \subseteq NL \), and thus \( RL \subseteq P \). The exercises ask you to show that \( BPL \subseteq P \) as well.

Now we describe the most famous randomized logspace algorithm, which is for the undirected \( s-t \)-connectivity problem.

\( u-s-t\text{-conn} = \{< G, s, t >: \text{nodes } s, t \text{ are connected in undirected graph } G \} \).

**Theorem 7.9** ([AKL\(^+\)79])

\( u-s-t\text{-conn} \in RL \).

\(^1\)The careful reader may have noted that when we defined nondeterministic logspace computation, we did not stipulate that every branch terminates in \( 2^{O(S(n))} \) time, whereas now we do. The reason is explored in the exercises.
Proof: (sketch) The proof uses properties of random walks on undirected graphs and we give an outline.

The algorithm consists of doing a random walk starting from \( s \) and running it for \( n^3 \) steps. At each step, the walk is at some vertex, say \( i \). It randomly picks an edge from all edges incident to \( i \) and goes to the other endpoint of this edge. (Notice, this requires it to choose among \( d_i \) possibilities where \( d_i \) is the degree of \( i \), which it can do using ideas similar to the proof of Claim 7.3.) If the walk reaches \( t \) in \( n^3 \) steps, accept, else reject.

The cover time of a random walk on a connected graph is the expected number of steps needed to visit all the vertices. It is known (see [Fei95] for the most recent bound) that the cover time on an \( n \)-node graph is at most \((1 + o(1))4n^3/27\). In our case, the “graph” on which the walk takes place is simply the connected component containing \( s \). If \( t \) is in this component, the probability that the walk has not reached it in \( n^3 \) steps is at most \( 4/27 \). If \( t \) is not in this component, the walk will never reach it. □

### 7.7.1 Universal traversal sequences

Universal traversal sequences represent an attempt to derandomize the algorithm in Theorem 7.9. If \( G \) is an undirected graph on \( n \) nodes and \( s \) is any node, then a a sequence \( u_1, u_2, \ldots, u_m \) where each \( u_i \in [1, n] \) can be viewed as a walk on the graph in the following way. If a node has \( d \) neighbors, then we number the outgoing edges from 1 to \( d \). At the \( i \)th step of the walk, we take the \( u_i \)th outgoing edge out of the current node if it exists; and otherwise stay at the current node.

**Definition 7.9 (Universal traversal sequence)** A sequence \( u_1, u_2, \ldots, u_m \) where each \( u_i \in [1, n] \) is a universal traversal sequence for \( n \)-node graphs if for every connected \( n \)-node graph \( G \) and every starting node \( s \), the above walk visits every node in the graph.

The exercises ask you to show that such sequences can be very short. The exercises in Chapter 18 give efficient constructions of fairly short sequences.

**Exercises**

§1 Show that for every \( c > 0 \), the following infinite sum is finite:

\[
\sum_{i \geq 1} \frac{c}{2^i}
\]
§2 Show, given input \([a]_2, [n]_2, [p]_2\), how to compute \(a^n \mod p\) in polynomial time. (Hint: use the binary representation of \(n\) and repeated squaring.)

§3 Let us study to what extent Claim 7.3 truly needs the assumption that \(\rho\) is efficiently computable. Describe a real number \(\rho\) such that given a random coin that comes up “Heads” with probability \(\rho\), a Turing machine can decide an undecidable language in polynomial time. (Hint: think of the real number \(\rho\) as an advice string. How can its bits be recovered?)

§4 Show that \(ZPP = RP \cap coRP\).

§5 A nondeterministic circuit has two inputs \(x, y\). We say that it accepts \(x\) iff there exists \(y\) such that \(C(x, y) = 1\). The size of the circuit is measured as a function of \(|x|\). Let \(NP/poly\) be the languages that are decided by polynomial size nondeterministic circuits. Show that \(BP \cdot NP \subseteq NP/poly\).

§6 Show using ideas similar to the Karp-Lipton theorem that if \(3\text{SAT} \in BP \cdot NP\) then \(PH\) collapses to \(\Sigma^p_3\). (Combined with above, this shows it is unlikely that \(3\text{SAT} \leq_r \overline{3\text{SAT}}\).)

§7 Show that \(BPL \subseteq P\) (Hint: use dynamic programming to compute the probability that the machine ends up in the accept configuration.)

§8 Show that there is a \(c, d > 1\) such that for all \(n\) there is a universal traversal sequence of size \(dn^c\). (Hint: Use Theorem 7.9 and the probabilistic method in a way similar to the way Theorem 7.5 was proved.)

§9 Show that the random walk idea of Theorem 7.9 does not work for directed graphs. In other words, describe a directed graph on \(n\) vertices and a starting point \(s\) such that the expected time to reach \(t\) is \(\Omega(2^n)\) even though there is a directed path from \(s\) to \(t\).

Chapter notes and history

Early researchers realized the power of randomization since their computations — e.g., for design of nuclear weapons — used probabilistic tools such as Monte Carlo simulations. Papers by von Neumann [von61] and de Leeuw et al. [LMSS56] describe probabilistic Turing machines. The definitions of
BPP, RP and ZPP are from Gill [Gil77]. (In an earlier conference paper [Gil74], Gill studies similar issues but seems to miss the point that a practical algorithm for deciding a language must feature a gap between the acceptance probability in the two cases.)

The algorithm used to show COMPOSITES is in RP is due to Solovay and Strassen [SS77]. Another primality test from the same era is due to Rabin [Rab80]. Over the years, better tests were proposed. In a recent breakthrough, Agrawal, Kayal and Saxena finally proved that COMPOSITES $\in$ P.

Lovász’s randomized NC algorithm [Lov79] for deciding the existence of perfect matchings is unsatisfying in the sense that when it outputs “Accept,” it gives no clue how to find a matching! Subsequent probabilistic NC algorithms can find a perfect matching as well; see [KUW86, MVV87].

BPP $\subseteq$ P/poly is from Adelman [Adl78]. BPP $\subseteq$ PH is due to Sipser [Sip83], and the stronger form BPP $\subseteq$ $\Sigma_2^p \cap \Pi_2^p$ is due to P. Gács. Recent work [] shows that BPP is contained in classes that are seemingly weaker than $\Sigma_2^p \cap \Pi_2^p$.

Even though a hierarchy theorem for BPP seems beyond our reach, a surprising result of Barak [Bar02] shows that a related class BPP/log (BPP machines with $O(\log n)$ bits of nonuniform advice) does have a hierarchy theorem. The proof is not elementary and uses advanced notions such as program checking (Chapter 10).

Readers interested in randomized algorithms are referred to the excellent book by Motwani and Raghavan [MR95] from the mid 1990s.
Chapter 8

Complexity classes having to do with counting

NEEDS ANOTHER PASS STILL

NP problems are defined using the notion of a certificate or witness for membership. In many contexts it is more natural to consider the number of certificates. This chapter studies \#P, a complexity class that captures this notion.

Counting problems arise in diverse fields, such as statistical estimation, statistical physics, network design, etc. Counting problems are also studied in a field of mathematics called enumerative combinatorics, which tries to obtain closed-form mathematical expressions for counting problems. To give an example, the number of spanning trees in a graph can be counted by means of a simple determinant computation. Results in this chapter will show that such efficiently computable expressions are unlikely to exist for a variety of simply stated counting problems.

Here is an example that suggests how counting problems can arise naturally in situations having to do with estimations of probability.

Example 8.1 In the GRAPH-RELIABILITY problem we are given a directed graph on \( n \) nodes. Suppose we are told that each node can fail with probability \( 1/2 \) and told to compute the probability that node 1 has a path to \( n \).

A moment’s thought shows that under this simple edge failure model, the remaining graph is uniformly chosen at random from all subgraphs of
the original graph. Thus the correct answer is
\[ \frac{1}{2^n} \text{(number of subgraphs in which node 1 has a path to } n) \]

We can view this as a *counting* version of the PATH problem.

In counting problems the output is a number, whereas thus far our definitions of complexity classes involved only decision problems. Therefore we first define the analog of \( P \) for function computations.

**Definition 8.1** \( FP \) is the set of all functions \( f : \{0, 1\}^* \to \mathbb{N} \) computable in polynomial time.

We will be interested in understanding which counting problems are in \( FP \).

**Definition 8.2** \#SAT is the problem of computing, given a boolean function \( \phi \), the number of satisfying assignments for \( \phi \).

\#CYCLE is the problem of computing, given a directed graph \( G \), the number of simple cycles in \( G \). (A simple cycle is one that does not visit any node twice.)

Clearly, if \#SAT \( \in \) FP then SAT \( \in \) P and so P = NP. Thus presumably #SAT \( \not\in \) FP. How about #CYCLE? The corresponding decision problem — given a directed graph decide if it has a cycle — can be solved in linear time by breadth-first-search. The next theorem suggests that the counting problem may be much harder.

**Theorem 8.1**
If #CYCLE \( \in \) FP, then P = NP.

**Proof:** We show that if #CYCLE can be computed in polynomial time, then HAMCYCLE \( \in \) P, where HAMCYCLE is the problem of deciding whether or not a digraph has a Hamiltonian cycle. Given a graph \( G \) with \( n \) nodes in the HAMCYCLE problem, we construct a graph \( G' \) for #CYCLE such that \( G \) has a HAMCYCLE iff \( G' \) has at least \( n^2 \) cycles.

To obtain \( G' \), replace each edge \((u, v)\) in \( G \) by a gadget as shown in Figure 8.1. The gadget has \( N = n \log_2 n + 1 \) levels. It is an acyclic digraph, so cycles in \( G' \) correspond to cycles in \( G \). Furthermore, there are \( 2^N \) directed
paths from \( u \) to \( v \) in the gadget, so a simple cycle of length \( l \) in \( G \) yields \((2^N)^l\) simple cycles in \( G' \).

Notice, if \( G \) has a Hamiltonian cycle, then \( G' \) has at least \((2^N)^n > n^2\) cycles. If \( G \) has no Hamiltonian cycle, then the longest cycle in \( G \) has length at most \( n - 1 \). The number of cycles is bounded above by \( n^{n-1} \). So \( G' \) can have at most \((2^N)^{n-1} \times n^{n-1} < n^{n^2} \) cycles. \( \Box \)

In the rest of the chapter we study \( \#P \), a complexity class that contains most counting problems. We also show a surprising connection between \( \text{PH} \) and \( \#P \), called \textit{Toda’s Theorem}. Along the way we encounter related classes such as \( \oplus P \).

### 8.1 \( \#P \) and \( \#P \)-completeness

First we define \( \#P \).

**Definition 8.3 \((\#P)\)** \( \#P \) is the set of all functions \( f : \{0,1\}^* \rightarrow \mathbb{N} \) such that there is a polynomial time NDTM \( M \) such that for all \( x \in \{0,1\}^* \),

\[
f(x) = |\{ y : \text{number of accepting branches in } M \text{'s computation tree on } x \}| \]

For example, \( \#\text{CYCLE} \in \#P \), because one can construct a polynomial time NDTM whose each branch guesses a sequence of nodes and accepts iff they form a simple cycle. In fact, all the problems mentioned in this chapter thus far are in \( \#P \).

Now we define \( \#P \)-completeness. Recall that in Chapter 4 we defined \textit{oracle} TMs. Thus \( FP^f \) is the set of functions that are computable by polynomial time TMs that have access to an oracle for function \( f \).

**Definition 8.4** A function \( f \) is \( \#P \)-complete if it is in \( \#P \) and every \( g \in \#P \) is also in \( FP^f \).

If \( f \in FP \) then \( FP^f = FP \). Thus the following is immediate.

**Proposition 8.2**

If \( f \) is \( \sharp \)-complete and \( f \in FP \) then \( P = NP \).
Counting versions of NP-complete languages naturally lead to #P-complete problems. This follows from the way reductions preserve the number of certificates (such a reduction is called *parsimonious*).

**Theorem 8.3**

#SAT is #P-complete

**Proof:** (sketch) Recall the Cook-Levin reduction of an NP language to SAT. It gives a one-one correspondence between the satisfying assignments of the boolean function produced and the accepting tableaux of the NP machine. Thus the ability to compute #SAT gives the ability to determine the number of accepting paths for the NP machine. □

### 8.1.1 Permanent and Valiant’s Theorem

Now we study another problem. The *permanent* of an \(n \times n\) matrix \(A\) is defined as

\[
\text{PERM}(A) = \sum_{\sigma \in S_n} \prod_{i=1}^{n} a_{\sigma(i)}
\]

where \(S_n\) denotes the set of all permutations of \(n\) elements. Recall that the expression for the determinant is similar

\[
\text{det}(A) = \sum_{\sigma \in S_n} (-1)^{\text{sgn}(\sigma)} \prod_{i=1}^{n} a_{\sigma(i)}
\]

except for an additional “sign” term. This similarity does not translate into computational equivalence: the determinant can be computed in polynomial time, whereas computing the permanent seems much harder, as we see below.

Now we try to understand the combinatorial meaning of the permanent function. First, suppose the matrix \(A\) has each entry in \(\{0, 1\}\). It may be viewed as the adjacency matrix of a bipartite graph \(G(X, Y, E)\), with \(X = \{x_1, \ldots, x_n\}\), \(Y = \{y_1, \ldots, y_n\}\) and \(\{x_i, y_j\} \in E\) iff \(A_{ij} = 1\). Then the term \(\prod_{i=1}^{n} a_{\sigma(i)}\) is 1 iff \(\sigma\) is a *perfect matching* (which is a set of \(n\) edges such that every node is in exactly one edge). Thus \(\text{PERM}(A)\) is simply the number of perfect matchings in \(G\). Thus \(\text{PERM}(\cdot) \in \#P\) for 0,1 matrices. If \(A\) is a \(-1, 0, 1\) matrix, then \(\text{PERM}(A) = |\{\sigma : \prod_{i=1}^{n} a_{\sigma(i)} = 1\}| - |\{\sigma : \prod_{i=1}^{n} a_{\sigma(i)} = -1\}|\), so one can make two calls to a #SAT oracle to compute \(\text{PERM}(A)\). Thus \(\text{PERM}(\cdot) \in \text{FP}^{\text{#SAT}}\) in this case. In fact one
8.1. #P AND #P-COMPLETENESS

Figure 8.2: This graph has permanent 0

can show for general integer matrices that computing the permanent is in FP\#SAT.

The next theorem came as a surprise to researchers in the 1970s, since it implies that if PERM ∈ FP then P = NP. Thus the permanent may be computationally much more difficult than the determinant.

**Theorem 8.4 (Valiant)**

PERM for 0, 1 matrices is #P-complete.

First we introduce an alternative way to look at the permanent. Consider matrix \( A \) as the adjacency matrix of a weighted \( n \)-node digraph (with possible self loops). Then the expression \( \prod_{i=1}^{n} a_{i\sigma(i)} \) is nonzero iff \( \sigma \) is a cycle-cover of \( A \) (A cycle cover is a subgraph in which each node has in-degree and out-degree 1; such a subgraph must be composed of cycles.) We define the weight of the cycle cover to be the product of the weights of the edges in it.

As a warm-up for the proof, we first use an example.

**Example 8.2** Consider the graph in Figure 8.2. (Unmarked edges have unit weight. We follow this convention throughout this chapter.) Even without knowing what the subgraph \( G' \) is, we show that the permanent of the whole graph is 0. For each cycle cover in \( G' \) there are exactly two cycle covers for the three nodes, one with weight 1 and one with weight \(-1\). Any non-zero weight cycle cover of the whole graph is composed of a cycle cover for \( G' \) and one of these two cycle covers. Thus the sum of the weights of all cycle covers of \( G \) is 0.
Now we prove Valiant’s Theorem.

**Proof:** We shall reduce the \#P-complete problem \#3SAT to PERM. Given a boolean formula $\phi$ with $n$ variables and $m$ clauses, first we shall show how to construct an integer matrix $A'$ with negative entries such that $\text{PERM}(A') = 4^m \cdot (#\phi)$. (#$\phi$ stands for the number of satisfying assignments of $\phi$). Later we shall show how to get a 0-1 matrix $A$ from $A'$ such that knowing PERM($A$) allows us to compute PERM($A'$).

The main idea is that there are two kinds of cycle covers in the digraph $G'$ associated with $A'$: those that correspond to satisfying assignments (we will make this precise) and those that don’t. Recall that PERM($A'$) is the sum of weights of all cycle covers of the associated digraph, where the weight of a cycle cover is the product of all edge weights in it. Since $A'$ has negative entries, some of these cycle covers may have negative weight. Cycle covers of negative weight are crucial in the reduction, since they help cancel out contributions from cycle covers that do not correspond to satisfying assignments. (The reasoning to prove this will be similar to that in Example 8.2.)

On the other hand, each satisfying assignment contributes $4^m$ to PERM($A'$), so $\text{PERM}(A') = 4^m \cdot (#\phi)$.

To construct $G'$ from $\phi$, we use three kinds of gadgets as shown in Figures 8.3, 8.4 and 8.5. There is a variable gadget per variable and a clause gadget per clause. There are two possible cycle covers of a variable gadget, corresponding to an assignment of 0 or 1 to that variable. Assigning 1 corresponds to a single cycle taking all the external edges (“true-edges”), and assigning 0 correspond to taking all the self-loops and taking the “false-edge”. Each external edge of a variable is associated with a clause in which the variable appears.
8.1. \#P AND \#P-COMPLETENESS

The clause gadget is such that the only possible cycle covers exclude at least one external edge. Also for a given (proper) subset of external edges used there is a unique cycle cover (of weight 1). Each external edge is associated with a variable appearing in the clause.

We will also use a graph called the XOR gadget (Figure 8.5) which has the following purpose: we want to ensure that exactly one of the edges $uu'$ and $vv'$ (see the schematic representation in Figure 8.5) is used in a cycle cover that contributes to the total count. So after inserting the gadget, we want to count only those cycle covers which either enter the gadget at $u$ and leave it at $u'$ or enter it at $v$ and leave it at $v'$. This is exactly what the gadget guarantees: one can check that the following cycle covers have total weight of 0: those that do not enter or leave the gadget; those that enter at $u$ and leave at $v'$, or those that enter at $v$ and leave at $u'$. In other words, the only cycle covers that have a nonzero contribution are those that either
Figure 8.6: For each clause and variable appearing in it, an XOR-gadget connects the corresponding external edges. There are $3m$ such connections in total.

(a) enter at $u$ and leave at $u'$ (which we refer to as using “edge” $uu'$) or (b) enter at $v$ and leave at $v'$ (referred to as using edge $vv'$). These are cycle covers in the “schematic graph” (which has edges as shown in Figure 8.5) which respect the XOR gadget.

The XOR gadgets are used to connect the variable gadgets to the corresponding clause gadgets so that only cycle covers corresponding to a satisfying assignment need be counted towards the total number of cycle covers. Consider a clause, and a variable appearing in it. Each has an external edge corresponding to the other, connected by an XOR gadget (figure 8.6). If the external edge in the clause is not taken (and XOR is respected) the external edge in the variable must be taken (and the variable is true). Since at least one external edge of each clause gadget has to be omitted, each cycle cover respecting all the XOR gadgets corresponds to a satisfying assignment. (If the XOR is not respected, we need not count such a cycle cover as its weight will be cancelled by another cover, as we argued above). Conversely, for each satisfying assignment, there is a cycle cover (unique, in the schematic graph) which respects all the XOR gadgets.

Now, consider a satisfying assignment and the corresponding cycle cover in the schematic graph. Passing (exactly one of) the external edges through the XOR gadget multiplies the weight of each such cover by 4. Since there
are \(3m\) XOR gadgets, corresponding to each satisfying assignment there are cycle covers with a total weight of \(4^{3m}\) (and all other cycle covers total to 0). So \(\text{PERM}(G') = 4^{3m} \#\phi\).

Finally we have to reduce finding \(\text{PERM}(G')\) to finding \(\text{PERM}(G)\), where \(G\) is an unweighted graph. First we reduce it to finding \(\text{PERM}(G)\) modulo \(2^N + 1\) for a large enough \(N\) (but still polynomial in \(|G'|\)). For this, we can replace -1 edges with edges of weight \(2^N\), which can be converted to \(N\) edges of weight 2 in series. Changing edges of (small) positive integral weights (i.e., multiple or parallel edges) to unweighted edges is as follows: cut each (repeated) edge into two and insert a node to connect them; add a self loop to the node. This does not change the permanent, and the new graph has only unweighted edges. □

8.1.2 Approximate solutions to \#P problems

Since computing exact solutions to \#P-complete problems is presumably difficult, one should try to compute approximate solutions in the sense of the following definition.

**Definition 8.5** Let \(f, g : \{0, 1\}^* \rightarrow \mathbb{N}\) be functions and \(c > 1\). We say that \(f\) approximates \(g\) within a factor \(c\) if for every string \(x\), \(g(x) \leq f(x) \leq c \cdot g(x)\).

Not all \#P problems behave identically with respect to this notion. Approximating certain problems within any constant factor \(c\) is NP-hard (see Exercises). For other problems such as 0/1 permanent, there is a **Fully polynomial randomized approximation scheme** (FPRAS), which is an algorithm which, for any \(\epsilon, \delta\), approximates the function within a factor \(1 + \epsilon\) (its answer may be incorrect with probability \(\delta\)) in time \(\text{poly}(n, \log 1/\delta, \log 1/\epsilon)\). This algorithm—as well as other similar algorithms for a host of \#P-complete problems—use the **Monte Carlo Markov Chain** technique. The result that spurred this development is due to Valiant and Vazirani and it shows that under fairly general conditions, approximately counting the number of elements in a set (membership in which is testable in polynomial time) is equivalent—in the sense that the problems are interreducible via polynomial-time randomized reductions—to the problem of generating a random sample from the set. We will not discuss this interesting area any further, though we will further explore the complexity of approximation in the exercises.
8.2 Toda’s Theorem: $\text{PH} \subseteq \text{P}^\#\text{SAT}$

Note that $\text{P} \subseteq \text{NP} \subseteq \text{P}^\#\text{SAT} \subseteq \text{PSPACE}$. An important question in the 1980s was the relative power of \text{PH} and \text{P}^\#\text{SAT}. Both are natural generalizations of \text{NP}, but it seemed that their features—alternation and the ability to count witnesses, respectively—are not directly comparable to each other. Thus it came as big surprise when in 1989 Toda showed:

**Theorem 8.5**
\[ \text{PH} \subseteq \text{P}^\#\text{SAT}. \]

Now we prove this result; this (circuit-based) proof is due to Kannan, Venkateswaran, Vinay, and Yao [KVVY93].

The following complexity class will be used in the proof.

**Definition 8.6** A language $L$ in the class $\oplus \text{P}$ (pronounced “parity P”) iff there exists a polynomial time NTM $M$ such that $x \in L$ iff the number of accepting paths of $M$ on input $x$ is odd.

**Definition 8.7** $\oplus \text{SAT} = \{\varphi : \varphi \text{ has an odd number of satisfying assignments}\}$.

A simple modification—essentially the same as the one in the proof of Theorem 8.3—of the Cook-Levin theorem shows that $\oplus \text{SAT}$ is complete for $\oplus \text{P}$. Now we give a characterization of $\oplus \text{P}$ using DC uniform circuits of exponential size; this is analogous to Theorem 6.11.

**Theorem 8.6**
$L \in \oplus \text{P}$ iff $L$ can be computed by a DC uniform circuit family $\{D_n\}$ that

- uses AND, OR, NOT, XOR gates.
- has size $2^{n \Theta(1)}$ and constant depth.
- XOR gates can have unbounded (exponential) fanin, but the AND, OR gates have fanin at most $n^{\Theta(1)}$.
- NOT gates can appear anywhere in the circuit.

**Proof:** Needs proof. □

We prove Toda’s Theorem in two steps. First we show that any $L \in \text{PH}$ can be randomly reduced to $\oplus \text{SAT}$ and then we show that this reduction can be transformed into a (deterministic) reduction to a $\#\text{P}$ problem. (Recall that randomized reductions between languages were defined in Definition 7.6.)
8.2. Toda’s Theorem: \( \text{PH} \subseteq P^{\#\text{SAT}} \)

8.2.1 Step 1: Randomized reduction from \( \text{PH} \) to \( \oplus \text{SAT} \)

This step consists of the following lemma.

**Lemma 8.7**

For every language \( L \in \text{PH} \) there is a probabilistic Turing machine \( M \) and \( c > 0 \) such that on input \( <1^n, r> \), where \( |r| \leq n^c \), \( M \) can compute the connection functions of an \( \oplus \ P \) circuit \( C_n \) of the type described in Theorem 8.6. Furthermore, for at least \( 2/3 \) of the \( r \), this circuit \( C_n \) has the property that it decides \( L \) on all inputs of size \( n \).

This Lemma will be used in the form of the following corollary, whose proof is straightforward since \( \oplus \text{SAT} \) is complete for \( \oplus \ P \).

**Corollary 8.8**

\( \forall L \in \text{PH}, \ L \leq_r \oplus \text{SAT} \).

To prove Lemma 8.7 we will need the Valiant-Vazirani lemma.

**Definition 8.8 (Nullspace)**

If \( a_1, \ldots, a_k \in \text{GF}(2)^N \), then

\[ (a_1, \ldots, a_k)^\perp = \{ s \in \text{GF}(2)^N : s \cdot a_1 = s \cdot a_2 = \ldots = s \cdot a_k = 0 \} \].

Let \( S \) be any nonempty subset of \( \text{GF}(2)^N \). The Valiant-Vazirani lemma gives a probabilistic way to pick a *unique* representative from \( S \), *and the method is the same for every subset \( S \): randomly pick \( i \in [0, N] \) and then \( a_1, a_2, \ldots, a_i \in \text{GF}(2)^N \); choose the vectors in \( S \cap (a_1, \ldots, a_i)^\perp \) as the representatives.*

The proof of the Lemma is left as an exercise.

**Lemma 8.9 (Valiant-Vazirani)**

If \( S \subseteq \text{GF}(2)^N \) is nonempty then there is an \( i \in \{0, 1, \ldots, N\} \) such that if \( a_1, \ldots, a_i \in \text{GF}(2)^N \) are chosen uniformly at random then

\[ \Pr[|S \cap (a_1, \ldots, a_i)^\perp| = 1] \geq \frac{1}{4}. \] (8.2)

**Remark 8.1** The Valiant-Vazirani lemma was proved as part of investigation of the complexity of the USAT problem, which involves finding a satisfying assignment of boolean formulae where the satisfying assignment (if one exists) is guaranteed to be unique. This problem arises in study of cryptographic primitives.

Now we prove a simple corollary that introduces the ideas used in the proof of Lemma 8.7.
CHAPTER 8. COMPLEXITY CLASSES HAVING TO DO WITH COUNTING

Corollary 8.10
\[ SAT \in R^{\oplus SAT}. \]

Proof: Let \( \varphi \) be some formula with \( N \) variables and \( S \) be its set of satisfying assignments. (If \( \varphi \) is unsatisfiable, \( S = \emptyset \).) The algorithm uses the deep fact that 1 is an odd number and 0 is even.

For any \( a_1, a_2, \ldots, a_i \in GF(2)^N \), we can write in polynomial time a formula \( \varphi_{a_1,\ldots,a_i} \) whose satisfying assignments are exactly the vectors in \( S \cap (a_1, \ldots, a_i)^\perp \). First, using the P-completeness of CIRCUIT-VALUE, construct a circuit which, given an input \( s \), accepts if \( s \) satisfies \( \varphi \) and \( s \cdot a_1 = s \cdot a_2 = \ldots = s \cdot a_i = 0 \). (The formula \( \varphi \) and vectors \( a_1, a_2, \ldots, a_n \) are “hardwired” into the circuit.) Then express the circuit as a boolean formula in the usual way.

The algorithm picks \( a_1, a_2, \ldots, a_N \in GF(2)^N \) randomly and for \( i = 1, 2, \ldots, N \) asks the \( \oplus SAT \) oracle if \( \varphi_{a_1,\ldots,a_i} \in \oplus SAT \). Every formula with an odd number of satisfying assignments is satisfiable, so if the oracle answers YES for any \( i \) then the algorithm accepts and otherwise it rejects. Clearly, the algorithm rejects unsatisfiable formulae with probability 1. By the VV Lemma it accepts satisfiable formulae with probability at least 1/4. \( \square \)

Now we prove Lemma 8.7 itself.

Proof: (Lemma 8.7) Let \( \{W_n\} \) be a DC uniform circuit family for \( L \) as described in Theorem 6.11 and \( M' \) be the associated polynomial-time Turing machine that computes its connection functions.

Let the size of circuit \( W_n \) be \( 2^N \) where \( N = n^{O(1)} \). The main idea is to transform \( W_n \) by replacing each OR and AND gate of \( W_n \) (recall their fanin may be up to \( 2^{n^{O(1)}} \)) by a depth 3 circuit using \( \oplus \) gates in which all AND, OR gates have fanin \( n^{O(1)} \). Doing this simultaneously for all gates gives the desired \( \oplus P \) circuit. Our transformations preserve the DC uniform property.

This gate conversion use the Valiant-Vazirani lemma, and, just like the proof of Corollary 8.10 uses the deep fact that 1 is an odd number and 0 is even. We describe the conversion of an OR gate.

For simplicity assume the OR gate is applied to \( 2^N \) wires in the circuit and we index them using vectors in \( GF(2)^N \): \( w_s \) is the wire for \( s \in GF(2)^N \). Let \( t \) be a string consisting of \( i \) picked randomly from \( \{0,1,\ldots,N \} \) and strings \( a_1, \ldots, a_i \) randomly from \( GF(2)^N \). Let \( R \) be a string consisting of \( 8N^2 \) independent samples of a type similar to \( t \).

Assume that the constants 0 and 1 are available as inputs; this is without loss of generality since \( 1 = x_1 \lor \overline{x_1} \) and \( 0 = x_1 \land \overline{x_1} \). First we change the OR
8.2. TODA’S THEOREM: $PH \subseteq P^{\#SAT}$

Figure 8.7: Probabilistic conversion of OR gate to $\oplus$ gate

into a circuit $g_t$ (see Figure 8.7) by replacing the the OR gate by $\oplus$ and the input $w_s$ by logical $AND$ of $w_s$ and the input 1 if $s \cdot a_1 = \ldots = s \cdot a_i = 0$ and $AND$ of $w_s$ and 0 otherwise. (The computation of whether to use 0 or 1 can be done in polynomial time using $r$, so the DC uniform property is maintained.) The following is just a restatement of the Valiant-Vazirani lemma.

**CLAIM 1:** For every sequence of values $\{w_s\}_{s \in GF(2)^N}$

$$\text{OR}(\{w_s\}) = 0 \implies \Pr_t[\text{OR}_t(\{w_s\}) = 1] = 0$$

$$\text{OR}(\{w_s\}) = 1 \implies \Pr_t[\text{OR}_t(\{w_s\}) = 1] \geq \frac{1}{4N}$$

Now the random string $R$ has $8N^2$ independent samples of strings like $t$, so define $g_t$ for each and take their OR; this is a depth 3 circuit called OR$_R$.

**CLAIM 2:** For every sequence of values $\{w_s\}_{s \in GF(2)^N}$:

$$\text{OR}(\{w_s\}) = 0 \implies \Pr_R[\text{OR}_R(\{w_s\}) = 1] = 0$$

$$\text{OR}(\{w_s\}) = 1 \implies \Pr_R[\text{OR}_R(\{w_s\}) = 1] \geq 1 - (1 - \frac{1}{4N})^{8N^2} \geq 1 - e^{-2N}$$
Apply this transformation to every gate in circuit $W_n$ to obtain the
⊕ P circuit $C_n$ (The AND gates are first transformed into OR’s using De-
Morgan’s law.) We emphasize that we the same random string $R$ for every
gate, thus using only $n^{O(1)}$ random bits total. Clearly, the connection func-
tions of $C_n$ can be computed using $R$ and the machine $M'$ that described
$W_n$.

**Claim 3:** For every input $x \in \{0,1\}^n$:
\[
\Pr_{R}[W_n(x) \neq C_n(x)] \leq 2^N e^{-2N}.
\]

To prove Claim 3 observe that fixing the input $x$ fixes, for each gate in
$C_n$, a sequence of values $\{w_s\}$ for the input wires to the gate. The gate
transformation fails on this sequence of inputs with probability at most
$e^{-2N}$. The Claim follows by the union bound applied to all $2^N$ gates.

Since there are only $2^n$ choices for $x$, we conclude that the probability
(over the choice of $R$) is at least $1 - 2^n 2^N e^{-2N} > 2/3$ that $C_n$ computes the
same function as $W_n$. □

### 8.2.2 Step 2: Making the reduction deterministic

We use the following Lemma.

**Lemma 8.11**

There is a (deterministic) polynomial-time transformation that, given $<
\psi, 1^m>$, outputs a formula $\psi^*$ such that
\[
\psi \in \oplus\text{SAT} \implies \#(\psi^*) \equiv -1 (mod \ 2^{m+1})
\]
\[
\psi \not\in \oplus\text{SAT} \implies \#(\psi^*) \equiv 0 (mod \ 2^{m+1}).
\]

Here $\#(\psi)$ is the number of satisfying assignments to formula $\psi$.

Assuming Lemma 8.11 we complete the proof of Toda’s Theorem. Let
$L \in PH$. We show that we can decide whether an input $x \in L$ by asking
the $\#\text{P}$ oracle a single question about the number of accepting paths of
a particular NP machine on input $x$. Recall that Corollary 8.8 gives a
randomized reduction from $L$ to $\oplus\text{SAT}$. Suppose this reduction uses a
random string $r$ of length $m$ and produces a boolean formula $\phi_r$ such that
\[
x \in L \implies \Pr_r[\phi_r \in \oplus\text{SAT}] \geq 2/3
\]
\[
x \not\in L \implies \Pr_r[\phi_r \in \oplus\text{SAT}] \leq 1/3
\] (8.3)
8.2. **Toda's Theorem:** \( \text{PH} \subseteq \text{P}^\#\text{SAT} \)

The NP machine, given input \( x \), guesses a choice for \( r \) and then computes \( \phi_r \). Then it uses the reduction of Lemma 8.11 (applied on the input \( < \phi_r, 1^m > \)) to produce a boolean formula \( \phi_r^* \). Then it guesses an assignment for \( \phi_r^* \) and accepts iff the assignment satisfies \( \phi_r^* \). Clearly,

\[
\#	ext{ accepting paths of } M \text{ on } x = \sum_r \#(\phi_r^*) \quad (8.4)
\]

where the last line follows from noticing that there are only \( 2^m \) random strings \( r \), which is less than \( \frac{1}{2} \cdot 2^{m+1} \). Thus knowing the number of accepting paths of the NP machine on input \( x \) allows us to decide —using (8.3)— whether on not \( x \) is in the PH language \( L \). This proves Toda’s Theorem.

To finish we prove Lemma 8.11.

**Proof:** (Lemma 8.11) For formulas \( \varphi(x_1, \ldots, x_k) \) and \( \tau(y_1, \ldots, y_t) \), let \( \varphi + \tau \) and \( \varphi \cdot \tau \) be shorthands respectively for the following formulae:

\[
(\varphi + \tau)(z, x_1, \ldots, x_k, y_1, \ldots, y_t) = (z \land \varphi(x_1, \ldots, x_k)) \lor (z \land \tau(y_1, \ldots, y_t)) \quad (8.7)
\]

\[
(\varphi \cdot \tau)(x_1, \ldots, x_k, y_1, \ldots, y_t) = \varphi(x_1, \ldots, x_k) \land \tau(y_1, \ldots, y_t) \quad (8.8)
\]

These formulae have size linear in the sizes of \( \varphi, \tau \) and satisfy:

\[
\#(\varphi + \tau) = \#(\varphi) + \#(\tau)
\]

\[
\#(\varphi \cdot \tau) = \#(\varphi) \cdot \#(\tau).
\]

Now consider the formula \( 4\tau^3 + 3\tau^4 \) (where \( \tau^3 \) for example is shorthand for \( \tau \cdot (\tau \cdot \tau) \)). One can easily check that

\[
\#(\tau) \equiv -1 (mod \ 2^{2^i}) \implies \#(4\tau^3 + 3\tau^4) \equiv -1 (mod \ 2^{2^i+1}) \quad (8.9)
\]

\[
\#(\tau) \equiv 0 (mod \ 2^{2^i}) \implies \#(4\tau^3 + 3\tau^4) \equiv 0 (mod \ 2^{2^i+1}). \quad (8.10)
\]

Returning to our proof, let \( \psi_0 = \psi \) and \( \psi_{i+1} = 4\psi_i^3 + 3\psi_i^4 \). Let

\[
\psi^* = \psi_{\lceil \log(m+1) \rceil}
\]

Repeated use of equations (8.9), (8.10) shows that if \( \#(\psi) \) is odd, then \( \#(\psi^*) \equiv -1 (mod \ 2^{m+1}) \) and if \( \#(\psi) \) is even, then \( \#(\psi^*) \equiv 0 (mod \ 2^{m+1}) \). Also, the size of \( \psi^* \) is only polynomially larger than size of \( \psi \).

\( \square \)
8.3 Open Problems

- What is the exact power of $\oplus$SAT and $\#SAT$?
- What is the average case complexity of $n \times n$ permanent modulo small prime, say 3 or 5? Note that for a prime $p > n$, random self reducibility of permanent implies that if permanent is hard to compute on at least one input then it is hard to compute on $1 - O(p/n)$ fraction of inputs, i.e. hard to compute on average (see Theorem 10.7).

Exercises

§1 Show that computing the permanent for matrices with integer entries is in $\text{FP}^\#\text{SAT}$.

§2 Prove Theorem 8.6.

§3 Prove the Valiant-Vazirani Lemma.

§4 Show that if there is a polynomial-time algorithm that approximates $\#\text{CYCLE}$ within a factor 2, then $\text{P} = \text{NP}$.

§5 Show that for every $g \in \#P$ and every $\epsilon > 0$, there is a function in $\text{FP}^{\Sigma_p^p}$ that approximates $g$ within a factor $1 + \epsilon$. (Hint: Use hashing and ideas similar to those in the proof of $\text{BPP} \subseteq \text{PH}$, where we also needed to estimate the size of a set of strings.) Thus assuming $\text{PH}$ doesn’t collapse to a finite level, approximation is easier than exact computation.

§6 Show that every for every language in $\text{AC}^0$ there is a depth 3 circuit of $n^{\text{poly}(\log n)}$ size that decides it on $1 - 1/\text{poly}(n)$ fraction of inputs and looks as follows: it has a single $\oplus$ gate at the top and the other gates are $\lor, \land$ of fanin at most $\text{poly}(\log n)$.

Chapter notes and history

The definition of $\#P$ as well as several interesting examples of $\#P$ problems appeared in Valiant’s seminal paper [Val79b]. The $\#P$-completeness of the permanent is from his other paper [Val79a]. Toda’s Theorem is proved in [Tod91].
8.3. OPEN PROBLEMS

For an introduction to FPRAS’s for computing approximations to many counting problems, see the relevant chapter in Vazirani [Vaz01] (an excellent resource on approximation algorithms in general).

Need to list some \#P-complete problems from physics.
Chapter 9

Cryptography

“Anyone who considers arithmetical methods of producing random digits is, of course, in a state of sin.”
John von Neumann, quoted by Knuth 1981

“From times immemorial, humanity has gotten frequent, often cruel, reminders that many things are easier to do than to reverse.”
L. Levin [Lev]

The importance of cryptography in today’s online world needs no introduction. Here we focus on the complexity issues that underlie this field.

The traditional task of cryptography was to allow two parties to encrypt their messages so that eavesdroppers gain no information about the message. (See Figure 9.1.) Encryption techniques have been invented throughout history. They had one common characteristic: sooner or later they were broken.

In the post NP-completeness era, a crucial new idea was presented [DH76]: the code-breaker should be thought of as a resource-bounded computational device. Hence the security of encryption schemes ought to be proved by showing that breaking the code is a computationally difficult task, and thus the code-breaker Turing machine requires hundreds of years (say).

Early researchers tried to base the security of encryption methods upon the (presumed) intractability of NP-complete problems. This effort has not succeeded to date, seemingly because NP-completeness concerns the intractability of worst-case instances whereas cryptography needs problems
that are intractable on most instances. After all, when we encrypt email, we require that decryption should be difficult for an eavesdropper for all (or almost all) messages, not just for a few messages. Thus the concept most useful in this chapter will be average-case complexity\(^1\). We will see a class of functions called one-way functions that are easy to compute but hard to invert for most inputs—they are alluded to in Levin’s quote above. The famous RSA function, invented in 1977, is a conjectured one-way function.

Furthermore, in the past two decades, cryptographers have taken on tasks above and beyond the basic task of encryption—from implementing digital cash to maintaining the privacy of individuals in public databases. (We survey some applications in Section 9.4.) Surprisingly, one-way functions, originally useful in encryption, have sufficed for almost all these new tasks. The most important ingredient in these developments turns out to be an answer to the question: “What is a random string and how can we generate one?” The complexity-theoretic answer to this question leads to the notion of a pseudorandom generator, which is a central object; see Section 9.2. This notion is very useful in itself and is also a template for several other key definitions in cryptography, including that of encryption (see Section 9.4).

**Private key versus public key:** Solutions to the encryption problem today come in two distinct flavors. In private-key cryptography, one assumes that the two (or more) parties participating in the protocol share a private “key”—namely, a statistically random string of modest size—that is not

\(^1\)A problem’s average-case and worst-case complexities can differ radically. For instance, 3-COLOR is NP-complete on general graphs, but on most \(n\)-node graphs is solvable in quadratic time or less. A deeper study of average case complexity appears in Chapter 16.
known to the eavesdropper. In public-key cryptography we drop this assumption. Instead, each party picks encryption and decryption functions from a published family. The encryption function is public — i.e., published and known to everybody including the eavesdropper — whereas the decryption function is known only to its designer. A famous public-key encryption scheme is based upon the famous RSA function of Example 9.3. Most topics described in this chapter are traditionally labeled private key cryptography.

9.1 Hard-on-average problems and one-way functions

A basic cryptographic primitive is a one-way function. Roughly speaking, this is a function $f$ that is easy to compute but hard to invert. Notice that if $f$ is not one-to-one, then the inverse $f^{-1}(x)$ may not be unique. In such cases “inverting” means that given $f(x)$ the algorithm is able to produce some preimage, namely, any element of $f^{-1}(f(x))$. We say that the function is one-way function if inversion is difficult for the “average” (or “many”) $x$. Now we define this formally; a discussion of this definition appears below in Section 9.1.1. A function family $(g_n)$ is a family of functions where $g_n$ takes $n$-bit inputs. It is polynomial-time computable if there is a polynomial-time TM that given an input $x$ computes $g_{|x|}(x)$.

DEFINITION 9.1 (One-way function) A family of functions \( \{f_n : \{0,1\}^n \rightarrow \{0,1\}^{m(n)} \} \) is $\epsilon(n)$ one-way with security $s(n)$ if it is polynomial-time computable and furthermore for every algorithm $A$ that runs in time $s(n)$,

$$\Pr_{x \in \{0,1\}^n}[A \text{ inverts } f_n(x)] \leq \epsilon(n).$$

(9.1)

Now we give a few examples and discuss the evidence that they are hard to invert “on average inputs.”

EXAMPLE 9.1 The first example is motivated by the fact that finding the prime factors of a given integer is the famous FACTORING problem, for which the best current algorithm has running time about $2^{O(n^{1/3})}$ (and even that bounds relies on the truth of some unproven conjectures in number theory). The hardest inputs for current algorithms appear to be of the type $x \cdot y$, where $x, y$ are random primes of roughly equal size.

\(^{2}\)Practically, this could be ensured with a face-to-face meeting that might occur long before the transmission of messages.
Here is a first attempt to define a one-way function using this observation. Let \( \{f_n\} \) be a family of functions where \( f_n : \{0, 1\}^n \times \{0, 1\}^n \rightarrow \{0, 1\}^{2n} \) is defined as \( f_n([x], [y]) = [x \cdot y] \). If \( x \) and \( y \) are primes—which by the Prime Number Theorem happens with probability \( \Theta(1/n^2) \)—then \( f_n \) seems hard to invert. It is widely believed that there are \( c > 1, f > 0 \) such that family \( f_n \) is \( (1 - 1/n^c) \)-one-way with security parameter \( 2^n f \).

An even harder version of the above function is obtained by using the existence of a randomized polynomial-time algorithm \( A \) (which we do not describe) that, given \( 1^n \), generates a random \( n \)-bit prime number. Suppose \( A \) uses \( m \) random bits, where \( m = \text{poly}(n) \). Then \( A \) may be seen as a (deterministic) mapping from \( m \)-bit strings to \( n \)-bit primes. Now let function \( \tilde{f}_m \) map \( (r_1, r_2) \) to \( [A(r_1) \cdot A(r_2)] \), where \( A(r_1), A(r_2) \) are the primes output by \( A \) using random strings \( r_1, r_2 \) respectively. This function seems hard to invert for almost all \( r_1, r_2 \). (Note that any inverse \( (r'_1, r'_2) \) for \( \tilde{f}_m(r_1, r_2) \) allows us to factor the integer \( A(r_1) \cdot A(r_2) \) since unique factorization implies that the prime pair \( A(r'_1), A(r'_2) \) must be the same as \( A(r_1), A(r_2) \).) It is widely conjectured that there are \( c > 1, f > 0 \) such that \( \tilde{f}_n \) is \( 1/n^c \)-one-way with security parameter \( 2^n f \).

The FACTORING problem, a mainstay of modern cryptography, is of course the inverse of multiplication. Who would have thought that the humble multiplication, taught to children in second grade, could be the source of such power? The next two examples also rely on elementary mathematical operations such as exponentiation, albeit with modular arithmetic.

**Example 9.2** Let \( p_1, p_2, \ldots \) be a sequence of primes where \( p_i \) has \( i \) bits. Let \( g_i \) be the generator of the group \( \mathbb{Z}_{p_i}^* \), the set of numbers that are nonzero mod \( p_i \). Then for every \( y \in 1, \ldots, p_i - 1 \), there is a unique \( x \in \{1, \ldots, p_i - 1\} \) such that

\[ g_i^x \equiv y \pmod{p_i}. \]

Then \( x \rightarrow g_i^x \pmod{p_i} \) is a permutation on \( 1, \ldots, p_i - 1 \) and is conjectured to be one-way. The inversion problem is called the DISCRETE LOG problem. We show below using random self-reducibility that if it is hard on worst-case inputs, then it is hard on average.
We list some more conjectured one-way functions.

**Example 9.3 RSA function.** Let $m = pq$ where $p, q$ are large random primes and $e$ be a random number coprime to $\phi(m) = (p - 1)(q - 1)$. Let $Z_m^*$ be the set of integers in $[1, \ldots, m]$ coprime to $m$. Then the function is defined to be $f_{p,q,e}(x) = x^e \pmod{m}$. This function is used in the famous RSA public-key cryptosystem.

**Rabin function.** For a composite number $m$, define $f_m(x) = x^2 \pmod{m}$. If we can invert this function on a $1/poly(\log m)$ fraction of inputs then we can factor $m$ in $poly(\log m)$ time (see exercises).

Both the RSA and Rabin functions are useful in public-key cryptography. They are examples of trapdoor one-way functions: if the factors of $m$ (the “trapdoor” information) are given as well then it is easy to invert the above functions. Trapdoor functions are fascinating objects but will not be studied further here.

**Random subset sum.** Let $m = 10n$. Let the inputs to $f$ be $n$ positive $m$-bit integers $a_1, a_2, \ldots, a_n$, and a subset $S$ of $\{1, 2, \ldots, n\}$. Its output is $(a_1, a_2, \ldots, a_n, \sum_{i \in S} a_i)$. Note that $f$ maps $n(m + 1)$-bit inputs to $nm + m$ bits.

When the inputs are randomly chosen, this function seems hard to invert. It is conjectured that there is $c > 1, d > 0$ such that this function is $1/n^c$-one-way with security $2^{n^d}$.

### 9.1.1 Discussion of the definition of one-way function

We will always assume that the the one-way function under consideration is such that the security parameter $s(n)$ is superpolynomial, i.e., larger than $n^k$ for every $k > 0$. The functions described earlier are actually believed to be one-way with a larger security parameter $2^{n^\epsilon}$ for some fixed $\epsilon > 0$.

Of greater interest is the error parameter $\epsilon(n)$, since it determines the fraction of inputs for which inversion is easy. Clearly, a continuum of values is possible, but two important cases to consider are (i) $\epsilon(n) = (1 - 1/n^c)$ for some fixed $c > 0$, in other words, the function is difficult to invert on at least $1/n^c$ fraction of inputs. Such a function is often called a weak one-way function. The simple one-way function $f_n$ of Example 9.1 is conjectured to
be of this type. (ii) \( \epsilon(n) < 1/n^k \) for every \( k > 1 \). Such a function is called a strong one-way function.

Yao showed that if weak one-way functions exist then so do strong one-way functions. We will prove this surprising theorem (actually, something close to it) in Chapter 17. We will not use it in this chapter, except as a justification for our intuition that strong one-way functions exist. (Another justification is of course the empirical observation that the candidate one-way functions mentioned above do seem appear difficult to invert on most inputs.)

### 9.1.2 Random self-reducibility

Roughly speaking, a problem is random-self-reducible if solving the problem on any input \( x \) reduces to solving the problem on a sequence of random inputs \( y_1, y_2, \ldots \), where each \( y_i \) is uniformly distributed among all inputs. To put it more intuitively, the worst-case can be reduced to the average case. Hence the problem is either easy on all inputs, or hard on most inputs. (In other words, we can exclude the possibility that problem is easy on almost all the inputs but not all.) If a function is one-way and also randomly self-reducible then it must be a strong one-way function. This is best illustrated with an example.

**Theorem 9.1**

Suppose \( A \) is an algorithm with running time \( t(n) \) that, given a prime \( p \), a generator \( g \) for \( \mathbb{Z}^*_p \), and an input \( g^x \pmod{p} \), manages to find \( x \) for \( \delta \) fraction of \( x \in \mathbb{Z}^*_p \). Then there is a randomized algorithm \( A' \) with running time \( O(\frac{1}{\delta \log 1/\epsilon}(t(n) + \text{poly}(n))) \) that solves DISCRETE LOG on every input with probability at least \( 1 - \epsilon \).

**Proof:** Suppose we are given \( y = g^x \pmod{p} \) and we are trying to find \( x \). Repeat the following trial \( O(1/(\delta \log 1/\epsilon)) \) times: “Randomly pick \( r \in \{0, 1, \ldots, p-2\} \) and use \( A \) to try to compute the logarithm of \( y \cdot g^r \pmod{p} \). Suppose \( A \) outputs \( z \). Check if \( g^{z-r} \pmod{p} \) is \( y \), and if so, output \( z - r \pmod{p-1} \) as the answer.”

The main observation is that if \( r \) is randomly chosen, then \( y \cdot g^r \pmod{p} \) is randomly distributed in \( \mathbb{Z}^*_p \) and hence the hypothesis implies that \( A \) has a \( \delta \) chance of finding its discrete log. After \( O(1/(\delta \log 1/\epsilon)) \) trials, the probability that \( A \) failed every time is at most \( \epsilon \). □

**Corollary 9.2**

If for any infinite sequence of primes \( p_1, p_2, \ldots \), DISCRETE LOG mod \( p_i \) is hard on worst-case \( x \in \mathbb{Z}^*_p \), then it is hard for almost all \( x \).
9.2. WHAT IS A RANDOM-ENOUGH STRING?

Later as part of the proof of Theorem 9.5 we give another example of random self-reducibility: linear functions over $GF(2)$.

9.2 What is a random-enough string?

Cryptography often becomes much easier if we have an abundant supply of random bits. Here is an example.

Example 9.4 (One-time pad) Suppose the message sender and receiver share a long string $r$ of random bits that is not available to eavesdroppers. Then secure communication is easy. To encode message $m \in \{0,1\}^n$, take the first $n$ bits of $r$, say the string $s$. Interpret both strings as vectors in $GF(2)^n$ and encrypt $m$ by the vector $m + s$. The receiver decrypts this message by adding $s$ to it (note that $s + s = 0$ in $GF(2)^n$). If $s$ is statistically random, then so is $m + s$. Hence the eavesdropper provably cannot obtain even a single bit of information about $m$ regardless of how much computational power he expends.

Note that reusing $s$ is a strict no-no (hence the name “one-time pad”). If the sender ever reuses $s$ to encrypt another message $m'$ then the eavesdropper can add the two vectors to obtain $(m + s) + (m' + s) = m + m'$, which is some nontrivial information about the two messages.

Of course, the one-time pad is just a modern version of the old idea of using “codebooks” with a new key prescribed for each day.

One-time pads are conceptually simple, but impractical to generate because sources of quality random bits (e.g., those based upon quantum phenomena) are often slow. Also, distributing one-time pads securely to users poses problems. Cryptography’s suggested solution to such problems is to use a pseudorandom generator. This is a deterministically computable function $g : \{0,1\}^n \rightarrow \{0,1\}^{cn}$ (for some $c > 1$) such that if $x \in \{0,1\}^n$ is randomly chosen, then $g(x)$ “looks” random. Thus so long as users have been provided a common $n$-bit random string, they can use the generator to produce $n^c$ “random looking” bits, which can be used to encrypt $n^{c-1}$ messages of length $n$. (In cryptography this is called a stream cipher.)

Clearly, at this point we need an answer to the question posed in the Section’s title! Philosophers and statisticians have long struggled with this question.
Example 9.5  What is a random-enough string? Here is Kolmogorov’s definition: A string of length \( n \) is random if no Turing machine whose description length is \(< 0.99n \) (say) outputs this string when started on an empty tape. This definition is not very useful in the complexity setting because checking if a string is random is undecidable.

Statisticians have also attempted definitions which boil down to checking if the string has the “right number” of patterns that one would expect by the laws of statistics, e.g. the number of times 11100 appears as a substring. (See Knuth Volume 3 for a comprehensive discussion.) Such definitions are too weak in the cryptographic setting.

9.2.1 Blum-Micali and Yao definitions

Now we introduce two complexity-theoretic definitions of pseudorandomness due to Blum-Micali and Yao in the early 1980s. Let \( y|_{\leq i} \) denotes the first \( i \) bits of string \( y \).

The Blum-Micali definition is motivated by the observation that one property (in fact, the defining property) of a statistically random sequence of bits \( y \) is that given \( y|_{\leq i} \), we cannot predict \( y_{i+1} \) with odds better than \( 50/50 \) regardless of the computational power available to us. Thus one could define a “pseudorandom” string by considering predictors that have limited computational resources, and to show that they cannot achieve odds much better than \( 50/50 \) in predicting \( y_{i+1} \) from \( y|_{\leq i} \). Of course, this definition has the shortcoming that any single finite string would be predictable for a trivial reason: it could be hardwired into the program of the predictor Turing machine. To get around this difficulty the Blum-Micali definition (and also Yao’s definition below) defines pseudorandomness for distributions of strings rather than for individual strings. Furthermore, the definition concerns an infinite sequence of distributions, one for each input size.

Definition 9.2 (Blum-Micali) Let \( \{g_n\} \) be a polynomial-time computable family of functions, where \( g_n : \{0,1\}^n \rightarrow \{0,1\}^m \) and \( m = m(n) > n \). We say the family is \((\epsilon(n),t(n))\)-unpredictable if for every probabilistic polynomial-time algorithm \( A \) that runs in time \( t(n) \) and every large enough input size \( n \),

\[
\Pr[A \text{ predicts } g(x)_{i+1} \text{ given } g(x)|_{\leq i}] \leq \frac{1}{2} + \epsilon(n),
\]
where the probability is over the choice of \( x \in \{0,1\}^n, i \in \{1, \ldots, n\} \), and the randomness used by \( A \).

If for every fixed \( k \), the family \( \{g_n\} \) is \((1/n^c, n^k)\)-unpredictable for every \( c > 1 \), then we say in short that it is unpredictable by polynomial-time algorithms.

**Remark 9.1** Allowing the tester to be an arbitrary polynomial-time machine makes perfect sense in a cryptographic setting where we wish to assume nothing about the adversary except an upperbound on her computational power.

Pseudorandom generators proposed in the pre-complexity era, such as the popular linear or quadratic congruential generators do not satisfy the Blum-Micali definition because bit-prediction can in fact be done in polynomial time (this is referred to in von Neumann’s quote at the start of the chapter).

Yao gave an alternative definition in which the tester machine is given access to the entire string at once. This definition implicitly sets up a test of randomness analogous to the more famous Turing test for intelligence (see Figure 9.2). The tester machine \( A \) is given a string \( y \in \{0,1\}^c \) that is produced in one of two ways: it is either drawn from the uniform distribution on \( \{0,1\}^c \) or generated by taking a random string \( x \in \{0,1\}^n \) and stretching it using a deterministic function \( g : \{0,1\}^n \to \{0,1\}^{nc} \). The tester is asked to output “1” if the string looks random to it and 0 otherwise. We say that \( g \) is a pseudorandom generator if no polynomial-time tester machine \( A \) has a great chance of being able to determine which of the two distributions the string came from.

**Definition 9.3 ([Yao82])** Let \( \{g_n\} \) be a polynomial-time computable family of functions, where \( g_n : \{0,1\}^n \to \{0,1\}^m \) and \( m = m(n) > n \). We say it is a \((\delta(n), s(n))\)-pseudorandom generator if for every probabilistic algorithm \( A \) running in time \( s(n) \) and for all large enough \( n \)

\[
|\Pr_{y \in \{0,1\}^c}[A(y) = 1] - \Pr_{x \in \{0,1\}^n}[A(g_n(x)) = 1]| \leq \delta(n). \tag{9.2}
\]

We call \( \delta(n) \) the distinguishing probability and \( s(n) \) the security parameter.

If for every \( c', k > 1 \), the family is \((1/n^{c'}, n^k)\)-pseudorandom then we say in short that it is a pseudorandom generator.
Figure 9.2: Yao’s definition: If \( c > 1 \) then \( g : \{0, 1\}^n \rightarrow \{0, 1\}^{nc} \) is a pseudorandom generator if no polynomial-time tester has a good chance of distinguishing between truly random strings of length \( nc \) and strings generated by applying \( g \) on random \( n \)-bit strings.

### 9.2.2 Equivalence of the two definitions

Yao showed that the above two definitions are equivalent — up to minor changes in the security parameter, a family is a pseudorandom generator iff it is (bitwise) unpredictable. The hybrid argument used in this proof has become a central idea of cryptography and complexity theory.

The nontrivial direction of the equivalence is to show that pseudorandomness of the Blum-Micali type implies pseudorandomness of the Yao type. Not surprisingly, this direction is also more important in a practical sense. Designing pseudorandom generators seems easier for the Blum-Micali definition — as illustrated by the Goldreich-Levin construction below — whereas Yao’s definition seems more powerful for applications since it allows the adversary unrestricted access to the pseudorandom string. Thus Yao’s theorem provides a bridge between what we can prove and what we need.

**Theorem 9.3 (Yao)**

Let \( g_n : \{0, 1\}^n \rightarrow \{0, 1\}^{N(n)} \) be a family of functions where \( N(n) = n^k \) for some \( k > 1 \).

If \( g_n \) is \((\epsilon(n)N(n)/2t(n))\)-unpredictable where \( t(n) \geq N(n)^2 \) then it is \((\epsilon(n), t(n))\)-pseudorandom.

Conversely, if \( g_n \) is \((\epsilon(n), t(n))\)-pseudorandom, then it is \((\epsilon(n), t(n))\)-unpredictable.

**Proof:** The converse part is trivial since a bit-prediction algorithm can in particular be used to distinguish \( g(x) \) from random strings of the same length. It is left to the reader.
9.2. WHAT IS A RANDOM-ENOUGH STRING?

We show the nontrivial first half. Let \( N \) be shorthand for \( N(n) \). Suppose \( g \) is not \((\epsilon(n), t(n))\)-pseudorandom, and \( A \) is a distinguishing algorithm that runs in \( t(n) \) time and satisfies:

\[
\left| \Pr_{x \in B_n}[A(g(x)) = 1] - \Pr_{y \in \{0,1\}^n}[A(y) = 1] \right| > \epsilon(n). \tag{9.3}
\]

By considering either \( A \) or the algorithm that is \( A \) with the answer flipped, we can assume that the \(|\cdot| | \) can be removed and in fact

\[
\Pr_{x \in B_n}[A(g(x)) = 1] - \Pr_{y \in \{0,1\}^n}[A(y) = 1] > \epsilon(n). \tag{9.4}
\]

Consider \( B \), the following bit-prediction algorithm. Let its input be \( g(x) \) where \( x \in \{0,1\}^n \) and \( i \in \{0, \ldots, N-1\} \) are chosen uniformly at random. B’s program is: “Pick bits \( u_{i+1}, u_{i+2}, \ldots, u_N \) randomly and run \( A \) on the input \( g(x) \) and \( u_{i+1} u_{i+2} \ldots u_N \). If \( A \) outputs 1, output \( u_{i+1} \) else output \( \overline{u_{i+1}} \).” Clearly, \( B \) runs in time less than \( t(n) + O(N(n)) < 2t(n) \). To complete the proof we show that \( B \) predicts \( g(x)_{i+1} \) correctly with probability at least \( \frac{1}{2} + \frac{\epsilon(n)}{N} \).

Consider a sequence of \( N+1 \) distributions \( D_0 \) through \( D_N \) defined as follows (in all cases, \( x \in \{0,1\}^n \) and \( u_1, u_2, \ldots, u_N \in \{0,1\} \) are assumed to be chosen randomly)

\[
D_0 = u_1 u_2 u_3 u_4 \cdots u_N \tag{9.5}
\]

\[
D_1 = g(x)_1 u_2 u_3 \cdots u_N \tag{9.6}
\]

\[
\vdots \quad \vdots \tag{9.7}
\]

\[
D_i = g(x)_{\leq i} u_{i+1} \cdots u_N \tag{9.8}
\]

\[
\vdots \quad \vdots \tag{9.9}
\]

\[
D_N = g(x)_1 g(x)_2 \cdots g(x)_N \tag{9.10}
\]

Furthermore, we denote by \( \overline{D_i} \) the distribution obtained from \( D_i \) by flipping the \( i \)th bit (i.e., replacing \( g(x)_i \) by \( \overline{g(x)_i} \)). If \( D \) is any of these \( 2(N+1) \) distributions then we denote \( \Pr_{y \in D}[A(y) = 1] \) by \( q(D) \). With this notation we rewrite (9.4) as

\[
q(D_N) - q(D_0) > \epsilon(n). \tag{9.11}
\]

Furthermore, in \( D_i \), the \((i+1)\)th bit is equally likely to be \( g(x)_{i+1} \) and \( \overline{g(x)_{i+1}} \), so

\[
q(D_i) = \frac{1}{2}(q(D_{i+1}) + q(\overline{D}_{i+1})), \tag{9.12}
\]
Now we analyze the probability that $B$ predicts $g(x)_{i+1}$ correctly. Since $i$ is picked randomly we have

$$\Pr_i[B \text{ correct}] = \frac{1}{N} \sum_{i=0}^{n-1} \frac{1}{2} \left( \Pr_{x,u}[B \text{'s guess for } g(x)_{i+1} \text{ is correct } | \ u_{i+1} = g(x)_{i+1}] + \Pr_{x,u}[B \text{'s guess for } g(x)_{i+1} \text{ is correct } | \ u_{i+1} = \overline{g(x)_{i+1}}] \right).$$

Since $B$’s guess is $u_{i+1}$ iff $A$ outputs 1 this is

$$= \frac{1}{2N} \sum_{i=0}^{N-1} \left( q(D_{i+1}) + 1 - q(\overline{D_{i+1}}) \right)$$

$$= \frac{1}{2} + \frac{1}{2N} \sum_{i=0}^{N-1} \left( q(D_{i+1}) - q(\overline{D_{i+1}}) \right)$$

From (9.12), $q(D_{i+1}) - q(\overline{D_{i+1}}) = 2(q(D_{i+1}) - q(D_i))$, so this becomes

$$= \frac{1}{2} + \frac{1}{2N} \sum_{i=0}^{N-1} 2(q(D_{i+1}) - q(D_i))$$

$$= \frac{1}{2} + \frac{1}{N} (q(D_N) - q(D_0))$$

$$> \frac{1}{2} + \frac{\epsilon(n)}{N}.$$

This finishes our proof. \(\square\)

9.3 One-way functions and pseudorandom number generators

Do pseudorandom generators exist? Surprisingly the answer (though we will not prove it in full generality) is that they do if and only if one-way functions exist.

**Theorem 9.4**

One-way functions exist iff pseudorandom generators do.

Since we had several plausible candidates for one-way functions in Section 9.1, this result helps us design pseudorandom generators using those
candidate one-way functions. If the pseudorandom generators are ever proved to be insecure, then the candidate one-way functions were in fact not one-way, and so we would obtain (among other things) efficient algorithms for FACTORING and DISCRETE LOG.

The “if” direction of Theorem 9.4 is trivial: if \( g \) is a pseudorandom generator then it must also be a one-way function since otherwise the algorithm that inverts \( g \) would be able to distinguish its outputs from random strings.

The “only if” direction is more difficult and involves using a one-way function to explicitly construct a pseudorandom generator. We will do this only for the special case of one-way functions that are permutations, namely, they map \( \{0, 1\}^n \) to \( \{0, 1\}^n \) in a one-to-one and onto fashion. As a first step, we describe the Goldreich-Levin theorem, which gives an easy way to produce one pseudorandom bit, and then describe how to produce \( n^c \) pseudorandom bits.

### 9.3.1 Goldreich-Levin hardcore bit

Let \( \{f_n\} \) be a one-way permutation where \( f_n : \{0, 1\}^n \to \{0, 1\}^n \). Clearly, the function \( g : \{0, 1\}^n \times \{0, 1\}^n \to \{0, 1\}^{2n} \) defined as \( g(x, r) = (f(x), r) \) is also a one-way permutation. Goldreich and Levin showed that given \( (f(x), r) \), it is difficult for a polynomial-time algorithm to predict \( x \odot r \), the scalar product of \( x \) and \( r \) (mod 2). Thus even though the string \( (f(x), r) \) in principle contains all the information required to extract \( (x, r) \), it is computationally difficult to extract even the single bit \( x \odot r \). This bit is called a hardcore bit for the permutation. Prior to the Goldreich-Levin result we knew of hardcore bits for some specific (conjectured) one-way permutations, not all.

**Theorem 9.5 (Goldreich, Levin '86)**

Suppose that \( \{f_n\} \) is a family of \( \epsilon(n) \)-one-way permutation with security \( s(n) \). Let \( S(n) = (\min \{ s(n), \frac{1}{\epsilon(n)} \})^{1/8} \) Then for all algorithms \( A \) running in time \( S(n) \)

\[
\Pr_{x, r \in \{0, 1\}^n}[A(f_n(x), r) = x \odot r] \leq \frac{1}{2} + O\left(\frac{1}{S(n)}\right). \tag{9.13}
\]

**Proof:** Suppose that some algorithm \( A \) can predict \( x \odot r \) with probability \( 1/2 + \delta \) in time \( t(n) \). We show how to invert \( f_n(x) \) for \( O(\delta) \) fraction of the inputs in \( O(n^3 t(n)/\delta^4) \) time, from which the theorem follows.
Claim 9.6

Suppose that

\[ \Pr_{x,r \in \{0,1\}^n}[A(f_n(x), r) = x \odot r] \geq \frac{1}{2} + \delta. \]  

(9.14)

Then for at least \( \delta \) fraction of \( x \)'s

\[ \Pr_{r \in \{0,1\}^n}[A(f_n(x), r) = x \odot r] \geq \frac{1}{2} + \frac{\delta}{2}. \]  

(9.15)

Proof: We use an averaging argument. Suppose that \( p \) is the fraction of \( x \)'s satisfying (9.15). We have \( p \cdot 1 + (1 - p)(1/2 + \delta/2) \geq 1/2 + \delta \). Solving this with respect to \( p \), we obtain

\[ p \geq \frac{\delta}{2(1/2 - \delta/2)} \geq \delta. \]

We design an inversion algorithm that given \( f_n(x) \), where \( x \in \{0,1\}^n \), will try to recover \( x \). It succeeds with high probability if \( x \) is such that (9.15) holds, in other words, for at least \( \delta \) fraction of \( x \). Note that the algorithm can always check the correctness of its answer, since it has \( f_n(x) \) available to it and it can apply \( f_n \) to its answer and see if \( f_n(x) \) is obtained.

Warmup: Reconstruction when the probability in (9.15) is \( \geq 3/4 + \delta \).

Let \( P \) be any program that computes some unknown linear function over \( GF(2)^n \) but errs on some inputs. Specifically, there is an unknown vector \( x \in GF(2)^n \) such that

\[ \Pr_{r}[P(r) = x \cdot r] = 3/4 + \delta. \]  

(9.16)

Then we show to add a simple “correction” procedure to turn \( P \) into a probabilistic program \( P' \) such that

\[ \forall r \quad \Pr_{r}[P'(r) = x \cdot r] \geq 1 - \frac{1}{n^2}. \]  

(9.17)

(Once we know how to compute \( x \cdot r \) for every \( r \) with high probability, it is easy to recover \( x \) bit-by-bit using the observation that if \( e_i \) is the \( n \)-bit vector that is 1 in the \( i \)th position and zero elsewhere then \( x \cdot e_i = a_i \), the \( i \)th bit of \( a \).)

“On input \( r \), repeat the following trial \( O(\log n/\delta^2) \) times. Pick \( y \) randomly from \( GF(2)^n \) and compute the bit \( P(r+y)+P(y) \). At the end, output the majority value.”
The main observation is that when \( y \) is randomly picked from \( GF(2)^n \)
then \( r+y \) and \( y \) are both randomly distributed in \( GF(2)^n \), and hence
the probability that \( P(r+y) \neq a \cdot (r+y) \) or \( P(y) \neq a \cdot y \) is at most
\( 2 \cdot (1/4 - \delta) = 1/2 - 2\delta \). Thus with probability at least \( 1/2 + 2\delta \), each trial
produces the correct bit. Then Chernoff bounds imply that probability is
at least \( 1 - 1/n^2 \) that the final majority is correct.

**General Case:**

The idea for the general case is very similar, the only difference being
that this time we want to pick \( r_1, \ldots, r_m \) so that we already “know” \( x \odot r_i \).
The preceding statement may appear ridiculous, since knowing the inner
variables \( Y \) variables

\[ Y \]s such that, as described above, we “know” \( x \odot Y_S \) for each
subset \( S \). This is why we can assume that we know \( x \odot Y_S \) for each subset \( S \).

The details of the rest of the algorithm are similar to before. Pick \( m \)
pairwise independent vectors \( Y_S \)’s such that, as described above, we “know”
\( x \odot Y_S \) for all \( S \). For each \( i = 1, 2, \ldots, n \), and each \( S \) run \( A \) on the input
\((f_n(x), Y_S \oplus e_i) \) (where \( Y_S \oplus e_i \) is \( Y_S \) with its \( i \)th entry flipped). Compute
the majority value of \( A(f_n(x), Y_S \oplus e_i) - x \odot Y_S \) among all \( S \)’s and use it as
your guess for \( x_i \).

Suppose \( x \in GF(2)^n \) satisfies (9.15). We will show that this algorithm
produces all \( n \) bits of \( x \) with probability at least \( 1/2 \). Fix \( i \). For each
\( i \), the guess for \( x_i \) is a majority of \( m \) bits. The expected number of bits
among these that agree with \( x_i \) is \( m(1/2 + \delta/2) \), so for the majority vote
to result in the incorrect answer it must be the case that the number of
incorrect values deviates from its expectation by more than \( m\delta/2 \). Now, we can bound the variance of this random variable and apply Chebyshev’s inequality (Lemma A.9 in the Appendix) to conclude that the probability of such a deviation is \( \leq \frac{4}{m\delta^2} \).

Here is the calculation using Chebyshev’s inequality. Let \( \xi_S \) denote the event that \( A \) produces the correct answer on \((f_n(x), Y_S \oplus e_i)\). Since \( x \) satisfies (9.15) and \( Y_S \oplus e_i \) is randomly distributed over \( GF(2)^n \), we have \( E(\xi_S) = 1/2 + \delta/2 \) and \( \text{Var}(\xi_S) = E(\xi_S)(1 - E(\xi_S)) < 1 \). Let \( \xi = \sum_S \xi_S \) denote the number of correct answers on a sample of size \( m \). By linearity of expectation, \( E[\xi] = \frac{1}{2} + \frac{\delta}{2} \) and \( \text{Var}(\xi) = \sum_S \text{Var}(\xi_S) < m \).

Now, by Chebyshev’s inequality, the probability that the majority vote is incorrect is at most \( \frac{4}{m\delta^2} \). Finally, setting \( m > \frac{8}{n\delta^2} \), the probability of guessing the \( i \)-th bit incorrectly is at most \( \frac{1}{2} \). By the union bound, the probability of guessing the whole word incorrectly is at most \( \frac{1}{2} \). Hence, for every \( x \) satisfying (9.15), we can find the preimage of \( f(x) \) with probability at least \( \frac{1}{2} \), which makes the overall probability of inversion at least \( \delta/2 \). The running time is about \( m^2n \times (\text{running time of } A) \), which is \( n^3\delta^4 \times t(n) \), as we had claimed. \( \square \)

### 9.3.2 Pseudorandom number generation

We saw that if \( f \) is a one-way permutation, then \( g(x, r) = (f(x), r, x \oplus r) \) is a pseudorandom generator that stretches \( 2n \) bits to \( 2n + 1 \) bits. Stretching to even more bits is easy too, as we now show. Let \( f^i(x) \) denote the \( i \)-th iterate of \( f \) on \( x \) (i.e., \( f(f(f(\cdots(f(x)))))) \) where \( f \) is applied \( i \) times.

**Theorem 9.7**

If \( f \) is a one-way permutation then \( g_N(x, r) = (r, x \oplus r, f(x) \oplus r, f^2(x) \oplus r, \ldots, f^N(x) \oplus r) \) is a pseudorandom generator for \( N = n^c \) for any constant \( c > 0 \).

**Proof:** Since any distinguishing machine could just reverse the string as a first step, it clearly suffices to show that the string \((r, f^N(x) \oplus r, f^{N-1}(x) \oplus r, \ldots, f(x) \oplus r, x \oplus r)\) looks pseudorandom. By Yao’s theorem (Theorem 9.3), it suffices to show the difficulty of bit-prediction. For contradiction’s sake, assume there is a PPT machine \( A \) such that when \( x, r \in \{0, 1\}^n \) and \( i \in \{1, \ldots, N\} \) are randomly chosen,

\[
\Pr[A \text{ predicts } f^i(x) \oplus r \text{ given } (r, f^N(x) \oplus r, f^{N-1}(x) \oplus r, \ldots, f^{i+1}(x) \oplus r)] \geq \frac{1}{2} + \epsilon.
\]
We describe an algorithm $B$ that given $f(z), r$ where $z, r \in \{0, 1\}^n$ are randomly chosen, predicts the hardcore bit $z \odot r$ with reasonable probability, which contradicts Theorem 9.5.

Algorithm $B$ picks $i \in \{1, \ldots, N\}$ randomly. Let $x \in \{0, 1\}^n$ be such that $f_i(x) = z$. There is of course no efficient way for $B$ to find $x$, but for any $l \geq 1$, $B$ can efficiently compute $f_{i+l}(x) \odot r, f_{i+l-1}(x) \odot r, \ldots, f_{i+1}(x) \odot r$ and uses it as input to $A$. By assumption, $A$ predicts $f_i(x) \odot r = z \odot r$ with good odds. Thus we have derived a contradiction to Theorem 9.5. □

9.4 Applications

Now we give some applications of the ideas introduced in the chapter.

9.4.1 Pseudorandom functions

Pseudorandom functions are a natural generalization of (and are easily constructed using) pseudorandom generators. This is a function $g : \{0, 1\}^m \times \{0, 1\}^n \to \{0, 1\}^m$. For each $K \in \{0, 1\}^m$ we denote by $g|_K$ the function from $\{0, 1\}^n$ to $\{0, 1\}^m$ defined by $g|_K(x) = g(K, x)$. Thus the family contains $2^m$ functions from $\{0, 1\}^n$ to $\{0, 1\}^m$, one for each $K$.

We say $g$ is a pseudorandom function generator if it passes a “Turing test” of randomness analogous to that in Yao’s definition of a pseudorandom generator (Definition 9.3).

Recall that the set of all functions from $\{0, 1\}^n$ to $\{0, 1\}^m$, denoted $\mathcal{F}_{n,m}$, has cardinality $(2^m)^{2^n}$. The PPT machine is presented with an “oracle” for a function from $\{0, 1\}^n$ to $\{0, 1\}^n$. The function is one of two types: either a function chosen randomly from $\mathcal{F}_{n,m}$, or a function $f|_K$ where $K \in \{0, 1\}^m$ is randomly chosen. The PPT machine is allowed to query the oracle in any points of its choosing. We say $f|_K$ is a pseudorandom function generator if for all $c > 1$ the PPT has probability less than $n^{-c}$ of detecting which of the two cases holds. (A completely formal definition would resemble Definition 9.1 and talk about a family of generators, one for each $n$. Then $m$ is some function of $n$.)

Now we describe a construction of a pseudorandom function generator $g$ from a length-doubling pseudorandom generator $f : \{0, 1\}^m \to \{0, 1\}^{2^m}$. For any $K \in \{0, 1\}^m$ let $T_K$ be a complete binary tree of depth $n$ whose each node is labelled with an $m$-bit string. The root is labelled $K$. If a node in the tree has label $y$ then its left child is labelled with the first $m$ bits of $y$. Then $g|_K$ can be constructed as follows: for each node $y$ in $T_K$ we define $g|_K(y)$ to be the function from $\{0, 1\}^n$ to $\{0, 1\}^m$ such that $g|_K(y)(x) = f_K(x \odot y)$.

We claim that $g|_K$ is a pseudorandom function generator. To see this, note that the PPT can only query the oracle at points in $\{0, 1\}^n$, and so cannot distinguish between a randomly chosen function from $\mathcal{F}_{n,m}$ and a function $f|_K$. □
Figure 9.3: Constructing a pseudorandom function from \( \{0,1\}^n \) to \( \{0,1\}^m \) using a random key \( K \in \{0,1\}^m \) and a length-doubling pseudorandom generator \( g : \{0,1\}^m \rightarrow \{0,1\}^{2^m} \).

\( f(y) \) and the right child is labelled with the last \( m \) bits of \( f(y) \). Now we define \( g(K,x) \). For any \( x \in \{0,1\}^n \) interpret \( x \) as a label for a path from root to leaf in \( T_K \) in the obvious way and output the label at the leaf. (See Figure 9.3.)

We leave it as an exercise to prove that this construction is correct.

A pseudorandom function generator is a way to turn a random string \( K \) into an implicit description of an exponentially larger “random looking” string, namely, the table of all values of the function \( g|_K \). This has proved a powerful primitive in cryptography; see the next section. Furthermore, pseudorandom function generators have also figured in a very interesting explanation of why current lowerbound techniques have been unable to separate P from NP; see Chapter 24.

### 9.4.2 Private-key encryption: definition of security

We hinted at a technique for private-key encryption in our discussion of a one-time pad (including the pseudorandom version) at the start of Section 9.2. But that discussion completely omitted what the design goals of the encryption scheme were. This is an important point: design of insecure systems often traces to a misunderstanding about the type of security ensured (or not ensured) by an underlying protocol.

The most basic type of security that a private-key encryption should ensure is semantic security. Informally speaking, this means that whatever can be computed from the encrypted message is also computable without access to the encrypted message and knowing only the length of the message. The formal definition is omitted here but it has to emphasize the facts...
that we are talking about an ensemble of encryption functions, one for each message size (as in Definition 9.1) and that the encryption and decryption is done by probabilistic algorithms that use a shared private key, and that for every message the guarantee of security holds with high probability with respect to the choice of this private key.

Now we describe an encryption scheme that is semantically secure. Let \( f : \{0,1\}^n \times \{0,1\}^n \rightarrow \{0,1\}^n \) be a pseudorandom function generator. The two parties share a secret random key \( K \in \{0,1\}^n \). When one of them wishes to send a message \( x \in \{0,1\}^n \) to the other, she picks a random string \( r \in \{0,1\}^n \) and transmits \((r, x \oplus f_K(r))\). To decrypt the other party computes \( f_K(r) \) and then XORs this string with the last \( n \) bits in the received text.

We leave it as an exercise to show that this scheme is semantically secure.

### 9.4.3 Derandomization

The existence of pseudorandom generators implies subexponential deterministic algorithms for \( \text{BPP} \): this is usually referred to as derandomization of \( \text{BPP} \). (In this case, the derandomization is only partial since it results in a subexponential deterministic algorithm. Stronger complexity assumptions imply a full derandomization of \( \text{BPP} \), as we will see in Chapter 18.)

**Theorem 9.8**

*If for every \( c > 1 \) there is a pseudorandom generator that is secure against circuits of size \( n^c \), then \( \text{BPP} \subseteq \cap_{\epsilon > 0} \text{DTIME}(2^{n^\epsilon}) \).*

**Proof:** Let us fix an \( \epsilon > 0 \) and show that \( \text{BPP} \subseteq \text{DTIME}(2^{n^\epsilon}) \).

Suppose that \( M \) is a \( \text{BPP} \) machine running in \( n^k \) time. We can build another probabilistic machine \( M' \) that takes \( n^\epsilon \) random bits, stretches them to \( n^k \) bits using the pseudorandom generator and then simulates \( M \) using this \( n^k \) bits as a random string. Obviously, \( M' \) can be simulated by going over all binary strings \( n^\epsilon \), running \( M' \) on each of them, and taking the majority vote.

It remains to prove that \( M \) and \( M' \) accept the same language. Suppose otherwise. Then there exists an infinite sequence of inputs \( x_1, \ldots, x_n, \ldots \) on which \( M \) distinguishes a truly random string from a pseudorandom string with a high probability, because for \( M \) and \( M' \) to produce different results, the probability of acceptance should drop from \( 2/3 \) to below \( 1/2 \). Hence we can build a distinguisher similar to the one described in the previous theorem by hardwiring these inputs into a circuit family. \( \square \)
The above theorem shows that the existence of hard problems implies that we can reduce the randomness requirement of algorithms. This “hardness versus randomness” tradeoff is studied more deeply in Chapter 18.

**Remark 9.2** There is an interesting connection to *discrepancy theory*, a field of mathematics. Let $\mathcal{S}$ be a set of subsets of $\{0,1\}^n$. Subset $A \subseteq \{0,1\}^n$ has discrepancy $\epsilon$ with respect to $\mathcal{S}$ if for every $s \in \mathcal{S}$,

$$\left| \frac{|s \cap A|}{|S|} - \frac{|A|}{2^n} \right| \leq \epsilon.$$ 

Our earlier result that $\text{BPP} \subseteq \text{P}/\text{poly}$ showed the existence of polynomial-size sets $A$ that have low discrepancy for all sets defined by polynomial-time Turing machines (we only described discrepancy for the universe $\{0,1\}^n$ but one can define it for all input sizes using lim sup). The goal of derandomization is to explicitly construct such sets; see Chapter 18.

### 9.4.4 Tossing coins over the phone and bit commitment

How can two parties $A$ and $B$ toss a fair random coin over the phone? (Many cryptographic protocols require this basic primitive.) If only one of them actually tosses a coin, there is nothing to prevent him from lying about the result. The following fix suggests itself: both players toss a coin and they take the XOR as the shared coin. Even if $B$ does not trust $A$ to use a fair coin, he knows that as long as his bit is random, the XOR is also random. Unfortunately, this idea also does not work because the player who reveals his bit first is at a disadvantage: the other player could just “adjust” his answer to get the desired final coin toss.

This problem is addressed by the following scheme, which assumes that $A$ and $B$ are polynomial time turing machines that cannot invert one-way permutations. The protocol itself is called *bit commitment*. First, $A$ chooses two strings $x_A$ and $r_A$ of length $n$ and sends a message $(f_n(x_A), r_A)$, where $f_n$ is a one-way permutation. This way, $A$ commits the string $x_A$ without revealing it. Now $B$ selects a random bit $b$ and conveys it. Then $A$ reveals $x_A$ and they agree to use the XOR of $b$ and $(x_A \oplus r_A)$ as their coin toss. Note that $B$ can verify that $x_A$ is the same as in the first message by applying $f_n$, therefore $A$ cannot change her mind after learning $B$’s bit. On the other hand, by the Goldreich–Levin theorem, $B$ cannot predict $x_A \oplus r_A$ from $A$’s first message, so this scheme is secure.
9.4. APPLICATIONS

9.4.5 Secure multiparty computations

This concerns a vast generalization of the setting in Section 9.4.4. There are \( k \) parties and the \( i \)th party holds a string \( x_i \in \{0,1\}^n \). They wish to compute \( f(x_1, x_2, \ldots, x_k) \) where \( f : \{0,1\}^{nk} \rightarrow \{0,1\} \) is a polynomial-time computable function known to all of them. (The setting in Section 9.4.4 is a subcase whereby each \( x_i \) is a bit —randomly chosen as it happens—and \( f \) is XOR.) Clearly, the parties can just exchange their inputs (suitably encrypted if need be so that unauthorized eavesdroppers learn nothing) and then each of them can compute \( f \) on his/her own. However, this leads to all of them knowing each other’s input, which may not be desirable in many situations. For instance, we may wish to compute statistics (such as the average) on the combination of several medical databases that are held by different hospitals. Strict privacy and nondisclosure laws may forbid hospitals from sharing information about individual patients. (The original example Yao gave in introducing the problem was of \( k \) people who wish to compute the average of their salaries without revealing their salaries to each other.)

We say that a multiparty protocol for computing \( f \) is secure if at the end no party learns anything new apart from the value of \( f(x_1, x_2, \ldots, x_k) \). The formal definition is inspired by the definition of a pseudorandom generator, and states that for each \( i \), the bits received by party \( i \) during the protocol should be computationally indistinguishable from completely random bits\(^3\).

It is completely nonobvious why such protocols must exist. Yao [Yao86] proved existence for \( k = 2 \) and Goldreich, Micali, Wigderson [GMW87] proved existence for general \( k \). We will not describe this protocol in any detail here except to mention that it involves “scrambling” the circuit that computes \( f \).

9.4.6 Lowerbounds for machine learning

In machine learning the goal is to learn a succinct function \( f : \{0,1\}^n \rightarrow \{0,1\} \) from a sequence of type \( (x_1, f(x_1)), (x_2, f(x_2)), \ldots, \) where the \( x_i \)'s are randomly-chosen inputs. Clearly, this is impossible in general since a random function has no succinct description. But suppose \( f \) has a succinct description, e.g. as a small circuit. Can we learn \( f \) in that case?

\(^3\)Returning to our medical database example, we see that the hospitals can indeed compute statistics on their combined databases without revealing any information to each other—at least any information that can be extracted feasibly. Nevertheless, it is unclear if current privacy laws allow hospitals to perform such secure multiparty protocols using patient data—an example of the law lagging behind scientific progress.
The existence of pseudorandom functions implies that even though a function may be polynomial-time computable, there is no way to learn it from examples in polynomial time. In fact it is possible to extend this impossibility result (though we do not attempt it) to more restricted function families such as NC^1 (see Kearns and Valiant [KV94]).

9.5 Recent developments

The earliest cryptosystems were designed using the SUBSET SUM problem. They were all shown to be insecure by the early 1980s. In the last few years, interest in such problems —and also the related problems of computing approximate solutions to the shortest and nearest lattice vector problems— has revived, thanks to a one-way function described in Ajtai [Ajt96], and a public-key cryptosystem described in Ajtai and Dwork [AD97] (and improved on since then by other researchers). These constructions are secure on most instances iff they are secure on worst-case instances. (The idea used is a variant of random self-reducibility.)

Also, there has been a lot of exploration of the exact notion of security that one needs for various cryptographic tasks. For instance, the notion of semantic security in Section 9.4.2 may seem quite strong, but researchers subsequently realized that it leaves open the possibility of some other kinds of attacks, including chosen ciphertext attacks, or attacks based upon concurrent execution of several copies of the protocol. Achieving security against such exotic attacks calls for many ideas, most notably zero knowledge (a brief introduction to this concept appears in Section 10.2.1).

Exercises

§1 Show that if P = NP then one-way functions and pseudorandom generators do not exist.

§2 (Requires just a little number theory). Prove that if some algorithm inverts the Rabin function \( f_m(x) = x^2 \mod m \) on a \( 1/\text{poly}(\log m) \) fraction of inputs then we can factor \( m \) in \( \text{poly}(\log m) \) time. (Hint: Suppose \( m = pq \) where \( p, q \) are prime numbers. Then \( x^2 \) has 4 “square roots” modulo \( m \).)

§3 Show that if \( f \) is a one-way permutation then so is \( f^k \) (namely, \( f(f(f(\cdots (f(x)))))) \) where \( f \) is applied \( k \) times) where \( k = n^c \) for some fixed \( c > 0 \).
§4 Assuming one-way functions exist, show that the above fails for one-way functions. (Hint: You have to design a one-way function where $f^k$ is not one-way.)

§5 Suppose $a \in \mathbf{GF}(2)^m$ is an unknown vector. Let $r_1, r_2, \ldots, r_m \in \mathbf{GF}(2)^m$ be randomly chosen, and $a \odot r_i$ revealed to us for all $i = 1, 2, \ldots, m$. Describe a deterministic algorithm to reconstruct $a$ from this information, and show that the probability (over the choice of the $r_i$’s) is at least $1/4$ that it works. (Hint: You need to show that a certain determinant is nonzero.) This shows that the “trick” in Goldreich-Levin’s proof is necessary.

§6 Suppose somebody holds an unknown $n$-bit vector $a$. Whenever you present a randomly chosen subset of indices $S \subseteq \{1, \ldots, n\}$, then with probability at least $1/2 + \epsilon$, she tells you the parity of the all the bits in $a$ indexed by $S$. Describe a guessing strategy that allows you to guess $a$ (an $n$ bit string!) with probability at least $(1 - \epsilon^2)^c$ for some constant $c > 0$.

§7 Suppose $g : \{0, 1\}^n \rightarrow \{0, 1\}^{n+1}$ is any pseudorandom generator. Then use $g$ to describe a pseudorandom generator that stretches $n$ bits to $n^k$ for any constant $k > 1$.

§8 Show the correctness of the pseudorandom function generator in Section 9.4.1. (Hint: Use a hybrid argument which replaces the labels on the first $k$ levels of the tree by completely random strings. Note that the random labels do not need to be assigned ahead of time — this would take at least $2^k$ time — but can be assigned on the fly whenever they are needed by the distinguishing algorithm.)

§9 Formalize the definition of semantic security and show that the encryption scheme in Section 9.4.2 is semantically secure. (Hint: First show that for all message pairs $x, y$ their encryptions are indistinguishable by polynomial-time algorithms. Why does this suffice?)

Chapter notes and history

In the 1940s, Shannon speculated about topics reminiscent of complexity-based cryptography. The first concrete proposal was made by Diffie and Hellman [DH76], though their cryptosystem was later broken. The invention of the RSA cryptosystem (named after its inventors Ron Rivest, Adi
Shamir, and Len Adleman) [RSA78] brought enormous attention to this topic. In 1981 Shamir [Sha83] suggested the idea of replacing a one-time pad by a pseudorandom string. He also exhibited a weak pseudorandom generator assuming the average-case intractability of the RSA function. The more famous papers of Blum and Micali [BM84] and then Yao [Yao82] laid the intellectual foundations of private-key cryptography. (The hybrid argument used by Yao is a stronger version of one in an earlier important manuscript of Goldwasser and Micali [GM84] that proposed probabilistic encryption schemes.) The construction of pseudorandom functions in Section 9.4.1 is due to Goldreich, Goldwasser, and Micali [GGM86]. The question about tossing coins over a telephone was raised in an influential paper of Blum [Blu82]. Today complexity-based cryptography is a vast field with several dedicated conferences. Goldreich [Gol04]’s two-volume book gives a definitive account.

A scholarly exposition of number theoretic algorithms (including generating random primes and factoring integers) appears in Bach and Shallit [BS96].

Theorem 9.4 and its very technical proof is in Håstad et al. [HILL99] (the relevant conference publications are a decade older).

Our proof of the Goldreich-Levin theorem is usually attributed to Rackoff (unpublished).
Chapter 10

Interactive proofs

“What is intuitively required from a theorem-proving procedure? First, that it is possible to “prove” a true theorem. Second, that it is impossible to “prove” a false theorem. Third, that communicating the proof should be efficient, in the following sense. It does not matter how long must the prover compute during the proving process, but it is essential that the computation required from the verifier is easy.”

Goldwasser, Micali, Rackoff 1985

Recall the certificate definition of NP. It consists of an interaction between a polynomial-time verifier and an all-powerful prover. For a language $L \in \text{NP}$, $x$ is in $L$ if and only if the prover can send $V$ a certificate (which may depend on $x$) that convinces the verifier. The certificate is analogous to a proof (for a mathematical theorem) written in a book, which the reader can read in isolation.

By contrast, when humans try to prove something to one another, they interact with one another. The verifier asks the prover for a series of explanations before he is convinced. This chapter formalizes such interactive proofs, a central notion of modern complexity theory. Intuitively, interaction should increase the class of languages for which the prover can give membership proofs to the verifier. This will indeed turn out to be the case.

Together with the related notion of zero knowledge, interactive proofs are also a cornerstone of complexity-based cryptography. We will address the cryptographic angle only briefly and concentrate instead on applications to complexity theory.

Studying interactive proofs allows us to give surprising new characterizations of conventional complexity classes such as PSPACE. We will also
prove useful results such as the fact that if PH does not collapse, then the famous graph isomorphism and approximate shortest vector problems are not NP-complete. We will also see interesting connections to the field of program checking, a small but influential research area. In fact, I feel that the field of interactive proofs will continue to yield surprising results for many more more years.

10.1 Warmup: Interactive proofs with a deterministic verifier

Let us consider what happens when we introduce interaction into the NP scenario above. Suppose we say that the language $L$ has an interactive proof system if there is a polynomial-time verifier $V$ that on $n$-bit inputs asks the prover a series of $N = \text{poly}(n)$ queries and receives answers to each. In general, the $(i + 1)$th query can depend upon the answers received to the first $i$ queries. The prover is any function that, given instance the input $x$ and the first $i$ queries $q_1, q_2, \ldots, q_i$, outputs a response $a_i$ to the $i$th query. For any prover $P$ we denote by $V^P(x)$ the final decision (i.e., 1 or 0) of $V$ on $x$ after interacting with $P$. We require

\begin{align}
   x \in L \Rightarrow \exists \text{ prover } P \text{ s.t. } V^P(x) = 1 \\
   x \notin L \Rightarrow \exists \text{ prover } P \text{ s.t. } V^P(x) = 0.
\end{align}

Unfortunately, interaction alone does not result in a new complexity class so long as the verifier is deterministic.

**Theorem 10.1**

Language $L$ has an interactive proof of the above type iff $L \in \text{NP}$.

**Proof:** Clearly, every NP language has a 1-round proof system. Now we prove that if a $L$ has an interactive proof system of this type then $L \in \text{NP}$. The certificate for membership is just the query-answer transcript $(q_1, a_1, q_2, a_2, \ldots, q_N, a_N)$ whose end-result is $V^P(x) = 1$. To verify this transcript, check that the first query of $V$ on $x$ is $q_1$, that the second query (after receiving $a_1$ as response to $q_1$) is $q_2$, and so on. Clearly, if such a transcript exists, it is proof that $x \in L$. □

---

1Understanding the complexity of these problems is a major open problem since neither a polynomial-time algorithm nor an NP-hardness result is known. Showing that the problems are unlikely to be NP-complete is therefore progress of sorts. The result for graph isomorphism is discussed in the exercises.
10.2 IP

In order to realize the full potential of interaction, we need to let the verifier be a probabilistic polynomial time TM. The set of languages which have interactive proof systems now jumps from NP to PSPACE, as we will see.

**Definition 10.1 (IP)** A language \( L \) is in IP if there is probabilistic, polynomial-time Turing Machine \( V \) that on an input \( x \) of size \( n \) can compute the following \( N = N(n) = n^{O(1)} \) functions \( q_1, q_2, \ldots, q_N \). Function \( q_i \) computes the \( i \)th query and its arguments are \( x \), a random string \( r \), and an \((i - 1)\)-tuple of strings \( (a_1, a_2, \ldots, a_{i-1}) \) (which are the prover’s answers to previous queries).

A prover \( P \) for inputs of length \( n \) is any arbitrary set of \( N \) functions where the \( i \)th function maps \( (x, q_1, q_2, \ldots, q_i) \) to an answer string \( a_i \).

\( V \) and \( P \) interact by sending queries and answers alternately (\( V \) goes first) and at the end of \( N \) rounds \( V \) outputs its 0/1 decision, denoted \( V^P(x, r) \).

The verifier has the property that:

\[
\begin{align*}
\text{if } x \in L & \Rightarrow \exists P \Pr_r[V^P(x, r) = 1] \geq 2/3 \quad (10.3) \\
\text{if } x \notin L & \Rightarrow \forall P \Pr_r[V^P(x, r) = 0] \geq 2/3. \quad (10.4)
\end{align*}
\]

If \( k \) is a fixed constant (independent of input size \( n \)) we define IP\([k]\) (for \( k \geq 2 \)) be the set of all languages that have a \( k \) round interactive proof similar to above, but with at most \( k \) rounds of interaction, where a “round” is either a query or a response and the first round is a query.

**Remark 10.1** We make a series of observations and leave some of the proofs as exercises.

1. Allowing the prover to be probabilistic (i.e., the answer function \( a_i \) depends upon some random string used by the prover) does not change the class IP. The reason is that for any language \( L \), if a probabilistic prover \( P \) results in making verifier \( V \) accept with some probability, then averaging implies there is a deterministic prover which makes \( V \) accept with the same probability.

2. Since the prover consists of arbitrary functions, it can in principle use infinite computational power (or even compute undecidable functions). However, one can show that given any verifier \( V \), we can compute the optimum prover (which, given \( x \), maximizes \( \Pr_r[V^P(x, r) = 1] \)) using \( \text{poly}(|x|) \) space. (Exercise!) Thus IP \( \subseteq \) PSPACE.
3. The probabilities of correctly classifying an input can be made arbitrarily large by using the same boosting technique we used for BPP: to replace $2/3$ by $1 - \exp(-m)$, sequentially repeat the protocol $m$ times and take the majority answer. Let us see that this works. If $x \in L$ then the same prover can be used on each repetition and in each of the independent trials it has a probability at least $2/3$ of making $V$ accept. Thus the chance is overwhelming that it succeeds for a majority of the trials. If $x \notin L$ then regardless of what strategy the prover uses in each repetition, the definition assures us that the probability that the verifier will not accept with probability more than $1/3$. Thus the probability that the verifier accepts in a majority of trials is tiny.

4. In fact, replacing the $2/3$ in () by 1 does not change the class IP. This is a nontrivial fact. It was originally proved in a complicated way but today can be proved using our characterization of IP later in Section 10.3.

5. We emphasize that the prover functions do not depend upon the verifier's random string. In other words, the verifier's random string is private. (Often these are called private coin interactive proofs.) Later we will also consider the model where the verifier's coins are public.

Let us look at an example of a language in IP.

**Example 10.1 (Graph non-isomorphism)** Recall that the usual ways of representing graphs—adjacency lists, adjacency matrices—involve a numbering of the vertices. We say two graphs $G_1$ and $G_2$ are isomorphic if they are the same up to a renumbering of vertices. In other words, if there is a permutation $\pi$ of the labels of the nodes of $G_1$ such that $\pi G_1 = G_2$. The graphs in figure 10.1, for example, are isomorphic with $\pi = (12)(3654)$. (That is, 1 and 2 are mapped to each other, 3 to 6, 6 to 5, 5 to 4 and 4 to 1.) If $G_1$ and $G_2$ are isomorphic, we write $G_1 \equiv G_2$. The **Graph Non-isomorphism** problem is this: given two graphs, decide if they not isomorphic?

The graph isomorphism problem is important in a variety of fields and has a rich history (see Hoffman [?]). It is clear that the complement of **Graph Non-isomorphism** is in **NP**— a certificate is simply the permutation $\pi$ that is an isomorphism. What is more surprising is that **Graph Non-isomorphism** is in **IP**. To show this, we give a private-coin protocol that satisfies definition 10.1:
10.2. IP

Figure 10.1: Two isomorphic graphs.

**Protocol: Private-coin Graph Non-isomorphism**

\[ V: \text{pick } i \in \{1, 2\} \text{ uniformly randomly. Randomly permute the vertices of } G_i \text{ to get a new graph } H. \text{ Send } H \text{ to } P. \]

\[ P: \text{identify which of } G_1, G_2 \text{ was used to produce } H. \text{ Let } G_j \text{ be that graph. Send } j \text{ to } V. \]

\[ V: \text{accept if } i = j; \text{ reject otherwise.} \]

To see that Definition 10.1 is satisfied by the above protocol, note that if \( G_1 \not\equiv G_2 \) then there exists a prover such that \( \Pr[V \text{ accepts}] = 1 \), because if the graphs are non-isomorphic, an all-powerful prover can certainly tell which one of the two is isomorphic to \( H \). On the other hand, if \( G_1 \equiv G_2 \) the best any prover can do is to randomly guess, because a random permutation of \( G_1 \) looks exactly like a random permutation of \( G_2 \). Thus \( \Pr[V \text{ accepts}] \leq 1/2. \)

10.2.1 Zero Knowledge and Cryptography

Our protocol for graph nonisomorphism has a special property: the verifier learns nothing new at the end of the interaction — except of course, that the graphs were nonisomorphic. Interactive proofs with this property are called zero-knowledge.

Let us try to formalize why the verifier learnt nothing. Suppose the verifier shows to a disinterested third party the transcript of everything that happened. The transcript shows that the verifier picked a random \( i \in \{1, 2\} \), randomly permuted the vertices of \( G_i \) to get \( H \), and then asked the prover
to identify $i$ such that $G_i \equiv H$. The prover responded with the correct $i$ (if the graphs were indeed nonisomorphic). Let us call such a transcript an accepting transcript, since the verifier accepts at the end of it.

We claim that such a transcript does not reveal any information about the prover. The reason is that the verifier could have easily (i.e., in polynomial time) generated the above transcript by itself, without any need for a prover, or even knowing that the graphs are indeed nonisomorphic! After all, it could have generated $i, H$ and then faked a response from the prover.

This is the essential feature of zero knowledge interactive proofs. The verifier’s questioning strategy is such that a randomized polynomial-time algorithm can generate accepting transcripts, without any need for interaction with a prover. (The formal definition is that the distribution produced by this generating algorithm is indistinguishable from the true distribution.)

One can see why such a concept might be useful in cryptography. It raises the possibility of parties being able to prove things to each other without revealing any secrets — e.g., to prove that you hold the password without revealing the password itself. This idea underlies a huge research effort in cryptography, but we will not study it further.

### 10.2.2 Public coins and AM

Allowing the prover full access to the verifier’s random string leads to the model of interactive proofs with public-coins.

**Definition 10.2 (AM, MA)** The classes $AM$ and $AM[k]$ are defined similarly\(^2\) to $IP$ and $IP[k]$ respectively, except the prover’s answers are allowed to depend upon the verifier’s random string $r$.

The classes $MA$ and $MA[k]$ are defined similarly, except the interaction starts with a message from the prover to the verifier.

The interactive proof for Graph Non-isomorphism seemed to crucially depend upon the fact that $P$ cannot see the random bits of $V$. If $P$ knew those bits, $P$ would know $i$ and so could trivially always guess correctly. The following result shows that the private coins were not really necessary!

**Theorem 10.2 (Goldwasser, Sipser)**

Graph Non-isomorphism $\in AM[2]$.

---

\(^2\)AM stands for “Arthur Merlin” and MA stands for “Merlin Arthur.” Arthur was a famous king of medieval England and Merlin was his court magician. Babai named these classes by drawing an analogy between the prover’s infinite power and Merlin’s magic.
10.2. IP

Proof: The key idea is to look at graph nonisomorphism in a different, more quantitative, way. (Aside: This is a good example of how nontrivial interactive proofs can be designed by recasting the problem.) Consider the set \( S = \{ H : H \equiv G_1 \text{ or } H \equiv G_2 \} \). The size of this set depends on whether \( G_1 \) is isomorphic to \( G_2 \). For a graph of \( n \) vertices, there are \( n! \) possible ways to label the vertices, so we have

\[
\begin{align*}
\text{if } G_1 \not\equiv G_2 & \text{ then } |S| = 2n! \\
\text{if } G_1 \equiv G_2 & \text{ then } |S| = n!
\end{align*}
\]

(Actually this intuition is not quite correct: we need the set
\[ S = \{ (H, \pi) : H \equiv G_1 \text{ or } H \equiv G_2 \text{ and } \pi \in \text{aut}(H) \}, \]
where \( \text{aut}(H) \) is the group of permutations \( \pi \) such that \( \pi(H) = H \).)

We can amplify the gap between the two cases. For \( k = 10 \), let \( U \) be the set of all \( k \)-tuples of (labeled) graphs and let \( S' \) be its subset that is the Cartesian product \( S^k = S \times S \times \cdots \times S \). Then \( |S'| \) is either \( 2^{10}(n!)^k \) or \( (n!)^k \).

For notational ease we choose to express this gap in a different way, letting \( m \) be such that \( 2^m \approx 10(n!)^k \) and noting that \( 2^{10} > 1000 \) obtain:

\[
\begin{align*}
\text{if } G_1 \not\equiv G_2 & \text{ then } |S'| \geq 100 \cdot 2^m \\
\text{if } G_1 \equiv G_2 & \text{ then } |S'| \leq \frac{1}{10^2} 2^m
\end{align*}
\]

Thus to convince the verifier that \( G_1 \not\equiv G_2 \), the prover has to convince the verifier that case (A) holds. This is done by the following set lowerbound protocol invented by Goldwasser and Sipser.

The idea is that the verifier randomly picks a hash function \( h \) from a family of 2-universal hash functions from \( U \) to \( \{0,1\}^m \). In case (A), with high probability, \( h(S') \approx \{0,1\}^m \), whereas in case (B), we have \( |h(S')| \leq |S'| \leq 2^m / 10 \). See Figure 10.2.

Lemma 10.3
Suppose \( S \geq \mu 2^m \) and \( \mathcal{H} \) is a pairwise-independent family of hash functions from \( U \) to \( \{0,1\}^m \) where \( S \subseteq U \). Then

\[
\Pr_{h \in \mathcal{H}} \left[ |h(S)| \geq \left( 1 - \frac{1}{\sqrt{\mu}} \right) 2^m \right] \geq 1 - \frac{1}{\sqrt{\mu}}.
\]

Proof: Left as exercise. □

This is the basis of our protocol: \( V \) randomly picks hash function \( h \in \mathcal{H} \) and \( y \in \{0,1\}^m \) and gives it to the prover. The prover has to respond
\begin{figure}
\centering
\begin{tikzpicture}
  \node (U) at (0,0) {$U$};
  \node (S) at (0,-1) {$S$};
  \node (h) at (1,0) {$h$};
  \node (01m) at (1,-1) {$\{0,1\}^m$};
  \draw[->] (h) to (01m);
  \draw[->] (U) to (S);
\end{tikzpicture}
\caption{The bigger $S'$ is, the more likely $h(S')$ will hit a given point in $\{0,1\}^m$.}
\end{figure}

with an $x \in S'$ such that $h(x) = y$, together with a proof (i.e., a $k$-tuple of isomorphisms to $G_1$ or $G_2$) that $x \in S'$. The verifier accepts iff the prover succeeds.

\begin{protocol}
\textbf{Protocol: Goldwasser-Sipser Set Lowerbound}
\begin{tabular}{ll}
V: & Randomly pick $h \in \mathcal{H}$, and $y \in R \{0,1\}^m$. Send $h,y$ to $P$. \\
P: & Try to find an $x \in S'$ such that $h(x) = y$. Send such an $x$ to $V$, together with a proof that $x \in S'$. \\
V: & If $x \in S'$ and $h(x) = y$, accept; otherwise reject.
\end{tabular}
\end{protocol}

It remains to be shown that the above protocol fits Definition 10.1. Suppose (B) holds, then as noted, $|h(S')| \leq 2^m/10$, so the probability that $P$ succeeds is at most $1/10$. If (A) holds, then Lemma 10.3 shows that

$$\Pr[y \in h(S')] \geq (1 - \frac{1}{\sqrt{100}})(1 - \frac{1}{\sqrt{100}}) = \frac{8}{10}.$$ 

This completes the proof that graph nonisomorphism has a 2-round public coins interactive proof. $\square$

\subsection{10.2.3 Some properties of IP and AM}

We state the following comments about IP[$\cdot$], AM[$\cdot$], without proof:

1. IP[$k$] $\subseteq$ AM[$k + 2$] for all constants $k$. This can be proved using the Goldwasser-Sipser technique.

2. Whereas BPP is a probabilistic version of P, AM[2] is as probabilistic version of NP. In fact AM[2] = BP·NP (see exercises) where BP·NP is the class in Definition 7.7. In particular it follows that AM[2] $\subseteq$ $\Sigma_3^p$. 
10.3. \( \text{IP} = \text{PSPACE} \)

Now we give surprising characterization of the set of languages that have interactive proofs\(^3\)

**Theorem 10.4 (LFKN, Shamir, 1990)**

\( \text{IP} = \text{PSPACE} \).

By our earlier Remark 10.1 we need only show that \( \text{PSPACE} \subseteq \text{IP}[\text{poly}(n)] \). To do so, we’ll show that \( \text{TQBF} \in \text{IP}[\text{poly}(n)] \). This is sufficient because every \( L \in \text{PSPACE} \) is polytime reducible to \( \text{TQBF} \). We note that our protocol for \( \text{TQBF} \) will use public coins and also has the property that if

---

\(^3\)Note that we tend to think of randomization as not adding much power to Turing machines. For instance, we will see in Chapter 18 that if certain complexity conjectures are true, then \( \text{P} = \text{BPP} \). Thus Theorem 10.4 often surprises students: it shows that randomization coupled with nondeterminism (i.e., the prover) is quite powerful. The analogy I sometimes use is painkillers and alcohol. Each is somewhat powerful in itself. The combination of the two is too powerful (and can be lethal).
the input is in TQBF then there is a prover which makes the verifier accept with probability 1.

Rather than tackle the job of designing a protocol for TQBF right away, let us first think about how to design one for 3SAT. How can the prover convince the verifier than a given 3CNF formula has no satisfying assignment? We show how to prove something even more general: the prover can prove to the verifier what the number of satisfying assignments is. (In other words, we will design a prover for #3SAT.) The idea of arithmetization introduced in this proof will also prove useful in our protocol for TQBF.

### 10.3.1 Arithmetization

The key idea will be to take an algebraic view of boolean formulae by representing them as polynomials. Note that 0, 1 can be thought of both as truth values and as elements of a finite field. Thus we have the following correspondence between formulas and polynomials when the variables take 0/1 values:

\[
\begin{align*}
  x \land y & \quad \leftrightarrow \quad X \cdot Y \\
  x \lor y & \quad \leftrightarrow \quad 1 - (1 - X)(1 - Y) \\
  \neg x & \quad \leftrightarrow \quad 1 - X \\
  x \lor y \lor \neg z & \quad \leftrightarrow \quad 1 - (1 - X)(1 - Y)Z
\end{align*}
\]

Note that if a 0/1 assignment (interpreted as an assignment of truth values) satisfies the boolean formula on the left then it makes the corresponding polynomial on the right evaluate to 1, and if it fails to satisfy the boolean formula then it makes the polynomial evaluate to 0.

Given any 3CNF formula \( \varphi(x_1, x_2, \ldots, x_n) \) with \( m \) clauses, we can write such a degree 3 polynomial for each clause. Multiplying these polynomials we obtain a degree 3\( m \) multivariate polynomial \( P_\varphi(X_1, X_2, \ldots, X_n) \) that evaluates to 1 for satisfying assignments and evaluates to 0 for unsatisfying assignments. (Note: in representing a polynomial we are allowed to keep use parenthesis, so \( P_\varphi(X_1, X_2, \ldots, X_n) \) has a representation of size \( O(m) \).) This conversion of \( \varphi \) to \( P_\varphi \) is called arithmetization. Once we have written such a polynomial, nothing stops us from going ahead and and evaluating the polynomial when the variables take arbitrary values from the field instead of just 0, 1. As we will see, this gives the verifier unexpected power over the prover.
10.3. Interactive protocol for $\#\text{SAT}_L$

To design a protocol for $\#\text{SAT}$ we give a protocol for $\#\text{3SAT}_L$, which is a decision version of the counting problem encountered in Chapter 8:

$\#\text{3SAT}_L = \{ \langle \phi, K \rangle : K \text{ is the number of satisfying assignments of } \phi \}$.

and $\phi$ is a 3CNF formula of $n$ variables and $m$ clauses.

**Theorem 10.5**
$\#\text{SAT}_L \in \text{IP}$.

**Proof:** Given input $\langle \phi, K \rangle$, we construct, by arithmetization, $P_{\phi}$. The number of satisfying assignments $\# \phi$ of $\phi$ is:

$$\# \phi = \sum_{b_1 \in \{0,1\}} \sum_{b_2 \in \{0,1\}} \cdots \sum_{b_n \in \{0,1\}} P_{\phi}(b_1, \ldots, b_n) \quad (10.5)$$

There is a general protocol, Sumcheck, for verifying equations such as (10.5).

**Sumcheck protocol.** Given a degree $d$ polynomial $g(X_1, \ldots, X_n)$ and an integer $K$, we present an interactive proof for the claim

$$K = \sum_{b_1 \in \{0,1\}} \sum_{b_2 \in \{0,1\}} \cdots \sum_{b_n \in \{0,1\}} g(X_1, \ldots, X_n). \quad (10.6)$$

$V$ simply needs to be able to evaluate $g$ for an arbitrary setting of values to the variables.

For each sequence of values $b_2, b_3, \ldots, b_n$ to $X_2, X_3, \ldots, X_n$, note that $g(X_1, b_2, b_3, \ldots, b_n)$ is a univariate degree $d$ polynomial. Thus the following is also a univariate degree $d$ polynomial:

$$h(X_1) = \sum_{b_2 \in \{0,1\}} \cdots \sum_{b_n \in \{0,1\}} g(X_1, b_2 \ldots, b_n).$$

If Claim (10.6) is true, then we have $h(0) + h(1) = K$.

To start, $V$ randomly picks a prime $p$ in the interval $[n^3, n^4]$ and instructs the prover to reduce all numbers modulo $p$ in the remaining protocol. (For instance, if the prover wants to send a polynomial, it only needs to send the coefficients modulo $p$.) All computations described below are also modulo $p$.

Consider the following protocol:
Protocol: Sumcheck protocol to check claim (10.6)

**V:** If \( n = 1 \) check that \( g(1) + g(0) = K \). If so accept, otherwise reject. If \( n \geq 2 \), ask \( P \) to send \( h(X_1) \) as defined above.

**P:** Sends some polynomial \( s(X_1) \) (may or may not be \( h(X_1) \)).

**V:** Reject if \( s(0) + s(1) \neq K \); otherwise pick a random \( a \). Recursively use the same protocol to check that

\[
s(a) = \sum_{b \in \{0,1\}} \cdots \sum_{b_n \in \{0,1\}} g(a, b_2, \ldots, b_n).
\]

If Claim (10.6) is true, the prover that always returns the correct polynomial will always convince \( V \). If (10.6) is false then we prove that \( V \) rejects with high probability:

\[
\Pr[V \text{ rejects } \langle K, g \rangle] \geq \left( 1 - \frac{d}{p} \right)^n.
\]  

(10.7)

With our choice of \( p \), the right hand side is about \( 1 - dn/p \), which is very close to 1 since \( p \geq n^3 \).

We prove (10.7) by induction on \( n \). The statement is true for \( n = 1 \) since \( V \) simply evaluates \( g(0), g(1) \) and rejects with probability 1 if their sum is not \( K \). Assume the hypothesis is true for degree \( d \) polynomials in \( n - 1 \) variables.

In the first round, the prover \( P \) is supposed to return the polynomial \( h \). If it indeed returns \( h \) then since \( h(0) + h(1) \neq K \) by assumption, \( V \) will immediately reject (i.e., with probability 1). So assume that the prover returns some \( s(X_1) \) different from \( h(X_1) \). Then there are at most \( d \) values \( a \) such that \( s(a) = h(a) \). Thus when \( V \) picks a random \( a \),

\[
\Pr_a[s(a) \neq h(a)] \geq 1 - \frac{d}{p}.
\]  

(10.8)

If \( s(a) \neq h(a) \) then the prover is left with an incorrect claim to prove in the recursive step. By the induction hypothesis, the prover fails to prove this false claim with probability at least \( \left( 1 - \frac{d}{p} \right)^{n-1} \). Thus we have

\[
\Pr[V \text{ rejects}] \geq \left( 1 - \frac{d}{p} \right) \cdot \left( 1 - \frac{d}{p} \right)^{n-1} = \left( 1 - \frac{d}{p} \right)^n
\]  

(10.9)
10.3. \( IP = PSPACE \)

This finishes the induction.

We still have to justify why it is OK to perform all operations modulo \( p \). The fear, of course, is that equation (10.6) might be false over the integers, but true mod \( p \), which happens if \( p \) divides the difference of the two sides of (10.6). The following lemma (encountered earlier in Example 7.5) implies that the chance that this happens for a random choice of \( p \) is small, since an \( n \)-bit integer (which is what \( K \) is) has at most \( n \) prime divisors.

**Lemma 10.6 (Fingerprinting)**

Suppose \( x, y, x \neq y \) are \( n \)-bit integers. Then there are at most \( n \) primes that divide \( |x - y| \).

\( \square \)

### 10.3.3 Protocol for TQBF: proof of Theorem 10.4

We use a very similar idea to obtain a protocol for TQBF. Given a quantified Boolean formula \( \Psi = \exists x_1 \forall x_2 \exists x_3 \cdots \forall x_n \phi(x_1, \ldots, x_n) \), we use arithmetization to construct the polynomial \( P_\phi \). We have that \( \Psi \in \text{TQBF} \) if and only if

\[
0 < \sum_{b_1 \in \{0, 1\}} \prod_{b_2 \in \{0, 1\}} \sum_{b_3 \in \{0, 1\}} \cdots \prod_{b_n \in \{0, 1\}} P_\phi(b_1, \ldots, b_n)
\]

A first thought is that we could use the same protocol as in the \#SATL case, except check that \( s(0) \cdot s(1) = K \) when you have a \( \prod \). But, alas, multiplication, unlike addition, increases the degree of the polynomial — after \( k \) steps, the degree could be \( 2^k \). Such polynomials may have \( 2^k \) coefficients and cannot even be transmitted in polynomial time if \( k \gg \log n \). The solution is to observe that the claimed statement only uses \( \{0, 1\} \) values. You can always replace a polynomial with a multi-linear function if you only evaluate it at \( \{0, 1\}^n \). Let \( R_{x_i} \) be a linearization operator defined as

\[
R_{x_1}[p(x_1, \ldots, x_n)] = (1 - x_1)p(0, x_2, \ldots, x_n) + (x_1)p(1, x_2, \ldots, x_n).
\] (10.10)

Then for any polynomial \( p \),

\[
R_{x_1}R_{x_2} \cdots R_{x_n}p(x_1, \ldots, x_n)
\]

is a multilinear polynomial that takes the same values as \( p \) whenever we substitute 0/1 values for the variables.

Now the idea is to work with the expression

\[
\exists x_1 R_{x_1} \forall x_2 R_{x_1} R_{x_2} \exists x_3 R_{x_1} R_{x_2} R_{x_3} \cdots \phi(x_1, \ldots, x_n)
\] (10.11)
The size of the expression is $O(1 + 2 + 3 + \cdots + n) = O(n^2)$. The protocol for $\#\text{SAT}_L$ can be used suitably modified, where in rounds involving the linearization operator, the verifier uses (10.10). The use of the linearization operator ensures that the polynomial which the prover needs to send at every round is linear, and hence the blowup in degrees is avoided. (Note that the verifier does not restrict attention to 0, 1 values; it substitutes arbitrary field elements for $x_1, x_2, \ldots$ as before.)

### 10.4 Interactive proof for the Permanent

Although the existence of an interactive proof for the Permanent follows from that for $\#\text{SAT}$ and TQBF, we describe a specialized protocol as well. This is both for historical context (this protocol was discovered before the other two protocols) and also because this protocol may be helpful for further research. (One example will appear in a later chapter.)

**Definition 10.3** Let $A \in F^{n \times n}$ be a matrix over the field $F$. The permanent of $A$ is:

$$perm(A) = \sum_{\sigma \in S_n} \prod_{i=1}^{n} a_{i, \sigma(i)}$$

The problem of calculating the determinant is $\#P$-complete (notice the similarity to the determinant which is polynomial time computable). Recall from previous lectures that $PH \subseteq P^{\text{PERM}}$ (Toda’s theorem).

Observation:

$$f(x_1, x_2, \ldots, x_n) := \text{perm} \begin{bmatrix} x_{1,1} & x_{1,2} & \cdots & x_{1,n} \\ x_{2,1} & \ddots & \cdots & x_{2,n} \\ \vdots & \ddots & \ddots & \vdots \\ x_{n,1} & x_{n,2} & \cdots & x_{n,n} \end{bmatrix}$$

is a degree $n$ polynomial since

$$f(x_1, x_2, \ldots, x_n) = \sum_{\sigma \in S_n} \prod_{i=1}^{n} x_{i, \sigma(i)}.$$

We now show two properties of the permanent problem. The first is ransom self reducibility, earlier encountered in Section 9.1.2:
Theorem 10.7 (Lipton ’88)
There is a randomized algorithm that, given an oracle that can compute the
permanent on $1 - \frac{1}{3n}$ fraction of the inputs in $F^{n \times n}$ (where the finite field $F$ has size $> 3n$), can compute the permanent on all inputs correctly with
high probability.

Proof: Let $A$ be some input matrix. Pick a random matrix $R \in R F^{n \times n}$
and let $B(x) := A + x \cdot R$ for a variable $x$. Notice that:

- $f(x) := \text{perm}(B)$ is a degree $n$ univariate polynomial.
- For any fixed $b \neq 0$, $B(b)$ is a random matrix, hence the probability
  that oracle computes $\text{perm}(B(b))$ correctly is at least $1 - \frac{1}{3n}$.

Now the algorithm for computing the permanent of $A$ is straightforward:
query oracle on all matrices $\{B(i) | 1 \leq i \leq n+1\}$. According to the union
bound, with probability of at least $1 - n + 1 \approx \frac{2}{3}$ the oracle will compute
the permanent correctly on all matrices.

Recall the fact (see Section A.2 in the Appendix) that given $n + 1$ (point,
value) pairs $\{(a_i, b_i) | i \in [n+1]\}$, there exists a unique a degree $n$ polynomial
$p$ that satisfies $\forall i \ p(a_i) = b_i$. Therefore, given that the values $B(i)$ are
correct, the algorithm can interpolate the polynomial $B(x)$ and compute
$B(0) = A$. □

Note: The above theorem can be strengthened to be based on the assump-
tion that the oracle can compute the permanent on a fraction of $\frac{1}{2} + \varepsilon$ for
any constant $\varepsilon > 0$ of the inputs. The observation is that not all values of
the polynomial must be correct for unique interpolation. See Chapter 23.

Another property of the permanent problem is downward self reducibility,
encountered earlier in context of SAT:

$$\text{perm}(A) = \sum_{i=1}^{n} a_{1,i}\text{perm}(A_{1,i}),$$

where $A_{1,i}$ is a $(n-1) \times (n-1)$ sub-matrix of $A$ obtained by removing the 1’st row
and i’th column of $A$ (recall the analogous formula for the determinant
uses alternating signs).

Definition 10.4 Define a $(n-1) \times (n-1)$ matrix $D_A(x)$, such that each
entry contains a degree $n$ polynomial. This polynomial is uniquely defined
by the values of the matrices $\{A_{1,i} | i \in [n]\}$. That is:

$$\forall i \in [n] \cdot D_A(i) = A_{1,i}$$
Where $D_A(i)$ is the matrix $D_A(x)$ with $i$ substituted for $x$. (notice that these equalities force $n$ points and values on them for each polynomial at a certain entry of $D_A(x)$, and hence according to the previously mentioned fact determine this polynomial uniquely)

Observation: $Perm(D_A(x))$ is a degree $n(n-1)$ polynomial in $x$.

We now show an interactive proof for the permanent (the decision problem is whether $perm(A) = k$ for some value $k$):

- **Round 1:** Prover sends to verifier a polynomial $g(x)$ of degree $n(n-1)$, which is supposedly $perm(D_A(x))$.

- **Round 2:** Verifier checks whether:
  \[ k = \sum_{i=1}^{m} a_{1,i} g(i) \]
  If not, rejects at once. Otherwise, verifier picks a random element of the field $b_1 \in_R F$ and asks the prover to prove that $g(b_1) = perm(D_A(b_1))$. This reduces the matrix dimension to $(n-2) \times (n-2)$.
  
  \[ \vdots \]

- **Round $2(n-1)-1$:** Prover sends to verifier a polynomial of degree 2, which is supposedly the permanent of a $2 \times 2$ matrix.

- **Round $2(n-1)$:** Verifier is left with a $2 \times 2$ matrix and calculates the permanent of this matrix and decides appropriately.

**Claim 10.8**

This is indeed an interactive protocol.

**Proof:** If $perm(A) = k$, then there exists a prover that makes the verifier accept with probability 1, this prover just returns the correct values of the polynomials according to definition.

On the other hand, suppose that $perm(A) \neq k$. If on the first round, the polynomial $g(x)$ sent is the correct polynomial $D_A(x)$, then:

\[ k \neq \sum_{i=1}^{m} a_{1,i} g(i) = perm(A) \]

And the verifier would reject. Hence $g(x) \neq D_A(x)$. According to the fact on polynomials stated above, these polynomials can agree on at most $n(n-1)$
10.5. THE POWER OF THE PROVER

A curious feature of many known interactive proof systems is that in order to prove membership in language $L$, the prover needs to do more powerful computation than just deciding membership in $L$. We give some examples.

1. The public coin system for graph nonisomorphism in Theorem 10.2 requires the prover to produce, for some randomly chosen hash function $h$ and a random element $y$ in the range of $h$, a $k$-tuple $x = (H_1, H_2, \ldots, H_k)$ of graphs where each $H_i$ is isomorphic to either $G_1$ or $G_2$, and $h(x) = y$. This seems harder than just solving graph nonisomorphism.

2. The interactive proof for $3\text{SAT}$, a language in coNP, requires the prover to do $\#P$ computations. (Recall that all of PH is in $P^{\#P}$.)

In both cases, it is an open problem whether the protocol can be redesigned to use a weaker prover.

Note that the protocol for TQBF is different in that the prover’s replies can be computed in PSPACE as well. This observation underlies the following result, which is in the same spirit as the Karp-Lipton results described in Chapter 6, except the conclusion is stronger since MA is contained in $\Sigma^P_2$ (see exercises).

Theorem 10.9

If $\text{PSPACE} \subseteq P/\text{poly}$ then $\text{PSPACE} = \text{MA}$.

Proof: If $\text{PSPACE} \subseteq P/\text{poly}$ then the prover in our TQBF protocol can be replaced by a circuit of polynomial size. Merlin (the prover) can just
give this circuit to Arthur (the verifier) in Round 1, who then runs the interactive proof using this “prover.” No more interaction is needed. Note that there is no need for Arthur to put blind trust in Merlin’s circuit, since the correctness proof of the TQBF protocol shows that if the formula is not true, then no prover can make Arthur accept with high probability. □

In fact, using the Karp-Lipton theorem one can prove a stronger statement (see Lemma 18.21).

10.6 Program Checking

The discovery of the interactive protocol for the permanent problem was triggered by a field called program checking. Blum and Kannan’s motivation for introducing this field was the fact that program verification (deciding whether or not a given program solves a certain computational task) is undecidable. They observed that in many cases we can guarantee a weaker guarantee of the program’s “correctness” on an instance by instance basis. This is encapsulated in the notion of a program checker. A checker for a program $P$ is itself another program that may run $P$ as a subroutine. Whenever $P$ is run on an input, $C$’s job is to detect if $P$’s answer is incorrect (“buggy”) on that particular instance. To do this, the checker may also compute $P$’s answer on some other inputs.

Program checking is sometimes also called instance checking, perhaps a more accurate name.

**Definition 10.5** Let $P$ be a claimed program for computational task $T$. A **checker** for $T$ is a probabilistic polynomial time TM, $C$, that, given any $x$, has the following behavior:

1. If $P$ is a correct program for $T$ (i.e., $\forall y P(y) = T(y)$), then $P[C^P \text{ accepts } P(x)] \geq \frac{2}{3}$

2. If $P(x) \neq T(x)$ then $P[C^P \text{ accepts } P(x)] < \frac{1}{3}$

Note that checkers are meant to catch bugs in the program. Even if the program is correct almost everywhere, any input $x$ such that $P(x) \neq T(x)$ will trigger the checker’s displeasure.

Surprisingly, for many problems, checking seems easier than actually computing the problem. (Blum and Kannan’s suggestion was to build checkers into the software whenever this is true; the overhead introduced by the checker would be negligible.)
Example 10.2 (Checker for Graph Non-Isomorphism) The input for the problem of Graph Non-Isomorphism is a pair of labelled graphs \(< G_1, G_2 >\), and the problem is to decide whether \(G_1 \equiv G_2\). As noted, we do not know of an efficient algorithm for this problem. But it has an efficient checker.

There are two types of inputs, depending upon whether or not the program claims \(G_1 \equiv G_2\). If it claims that \(G_1 \equiv G_2\) then one can change the graph little by little and use the program to verify this claim. (Left as an exercise.) We now show how to check the claim that \(G_1 \not\equiv G_2\) using our earlier interactive proof of Graph non-isomorphism.

Recall the IP for Graph Non-Isomorphism:

- In case prover admits \(G_1 \not\equiv G_2\) repeat \(k\) times:
  - Choose \(i \in_R \{1, 2\}\). Permute \(G_i\) randomly into \(H\)
  - Ask the prover \(< G_1, H >; < G_2, H >\) and check to see if the prover’s first answer is consistent.

Given a computer program that supposedly computes graph isomorphism, \(P\), how would we check its correctness? The program checking approach suggests to use an IP while regarding the program as the prover. Let \(C\) be a program that performs the above protocol with \(P\) as the prover, then:

Theorem 10.10

If \(P\) is a correct program for Graph Non-Isomorphism then \(C\) outputs "correct" always. Otherwise, if \(P(G_1, G_2)\) is incorrect then \(P\) outputs "correct" \(\leq 2^{-k}\). Moreover, \(C\) runs in polynomial time.

10.6.1 Languages that have checkers

The following theorem are a direct consequence of the IPs we have seen:

Theorem 10.11

The problems Graph Non-Isomorphism, Quadratic Residue, Permanent have checkers.

The following theorem shows that program checking is a robust notion:
Theorem 10.12 (attributed to Beigel in [BK95])

Let $\mathcal{F}$ be a time complexity class which is closed under multiplication, composition and addition of its time complexity functions (such as $P$ and $\text{TIME}(\log^O(1) n)$). Let $\pi_1$ and $\pi_2$ two problems reducible to each other in $\mathcal{F}$, and $C_1$ a checker for $\pi_1$ in $\mathcal{F}$. Then there is a checker $C_2$ for $\pi_2$ in $\mathcal{F}$.

Proof: (sketch) Basically one can produce the new checker using the checker for the first problem composed with a program that computes the reduction function. Correctness follows directly from the definition of reductions and the fact that the complexity class is closed under basic operations. □

Using this theorem, and the fact that P-complete languages are NC-reducible to each other, it suffices to show a checker in NC for one P-complete language (as was shown by Blum & Kannan) to obtain the following interesting fact:

Theorem 10.13

For any P-complete language there exists a program checker in NC

Since we believe that P-complete languages cannot be computed in NC, this provides strong evidence that checking is easier than actual computation.

The relation between program checking and interactive proofs is more precisely stated by the following theorem:

Definition 10.6 Let function restricted IP the set of all decision problems $\pi$ for which there exists an IP, where the prover is further restricted to be a function from the program instances to $\{\text{YES}/\text{NO}\}$.

In particular, the verifier may only ask questions that are instances of $\pi$ and the prover’s answers are history-independent: the answer to each question is the same regardless of when it is asked during the protocol.

Theorem 10.14

A language $L$ has a checker $\iff L \in \text{function restricted IP} \cap \text{co-function restricted IP}$.

Proof: $\Leftarrow$ Suppose that $L$ has an IP of one of the sorts above. A checker for the language would simulate the IP, using the supposed machine for the language (or co-language) as provers.

$\Rightarrow$ Suppose the language $L$ has a checker, denoted $M$. Then a (co-) function restricted IP for this language is just the simulation of the machine, such that each activation of $M$ of the oracle is a question for the prover. Correctness follows simply from definitions. □
10.7 Multiprover interactive proofs (MIP)

It is also possible to define interactive proofs that involve more than one prover. The important assumption is that the provers do not communicate with each other during the protocol. They may communicate before the protocol starts, and in particular, agree upon a shared strategy for answering questions. (The analogy often given is that of the police interrogating two suspects in separate rooms. The suspects may be accomplices who have decided upon a common story to tell the police, but since they are interrogated separately they may inadvertently reveal an inconsistency in the story.)

The set of languages with multiprover interactive provers is call MIP. The formal definition is analogous to Definition 10.1. We assume there are two provers (though one can also study the case of polynomially many provers; see the exercises), and in each round the verifier sends a query to each of them—the two queries need not be the same. Each prover sends a response in each round.

Clearly, IP ⊆ MIP. It is difficult to see why MIP should be any bigger than IP; in other words, to see how allowing an extra prover could help by much. The next theorem is therefore surprising (assuming PSPACE is a proper subset of NEXPTIME). We will outline a proof of this theorem in Chapter 19.

Theorem 10.15

\[ MIP = \text{NEXPTIME}. \]

Exercises

§1 Prove the assertions in Remark 10.1.

§2 We say integer \( y \) is a quadratic residue modulo \( m \) if there is an integer \( x \) such that \( y \equiv x^2 \pmod{m} \). Show that the following language is in IP[2]:

\[ \text{QUADRATIC-NONRESIDUOSITY} = \{(y,m) : y \text{ is not a quadratic residue modulo } m\}. \]

§3 Show that AM[2] = BP · NP

§4 Use the previous question to show that if graph isomorphism is NP-complete then \( \text{PH} = \Sigma^p_3 \). (Hint: use one of the problems in Chapter 7.)

§5 Prove Lemma 10.3. (Hint: use pairwise independence.)
§6 Show that $\text{MA} \subseteq \Sigma^p_2$.

§7 [BFNW93] Show that if $\text{EXPTIME} \subseteq \text{P/poly}$ then $\text{EXPTIME} = \text{MA}$. (Hint: The interactive proof for TQBF requires a prover that is a PSPACE machine.)

§8 Complete the proof in Example 10.2 that graph nonisomorphism has a checker. Specifically, you have to show that if the program claims that $G_1 \equiv G_2$ then we can do some further investigation (including calling the programs on other inputs) and with high probability conclude that either (a) conclude that the program was right on this input or (b) the program is wrong on some input and hence is not a correct program for graph nonisomorphism.

§9 Show that $\text{MIP} \subseteq \text{NEXPTIME}$.

§10 Show that if we redefine multiprover interactive proofs to allow, instead of two provers, as many as $m(n) = \text{poly}(n)$ provers on inputs of size $n$, then the class $\text{MIP}$ is unchanged. (Hint: Show how to simulate $\text{poly}(n)$ provers using two. In this simulation, one of the provers plays the role of all $m(n)$ provers, and the other prover is asked to simulate one of the provers, chosen randomly from among the $m(n)$ provers. Then repeat this a few times.)

Chapter notes and history

Interactive proofs were defined in 1985 by Goldwasser, Micali, Rackoff [GMR89] for cryptographic applications and (independently, and using the public coin definition) by Babai and Moran [BM88]. Simulations of private coins by public coins we given by Goldwasser and Sipser [GS87]. The general feeling at the time was that interactive proofs are only a “slight” extension of NP and that not even $\text{3SAT}$ has interactive proofs. The result $\text{IP} = \text{PSPACE}$ was a big surprise, and the story of its discovery is very interesting.

In the late 1980s, Blum and Kannan [BK95] introduced the notion of program checking. Around the same time, manuscripts of Beaver and Feigenbaum [BF90] and Lipton [Lip91] appeared. Inspired by some of these developments, Nisan proved in December 1989 that $\#\text{SAT}$ has multiprover interactive proofs. He announced his proof in an email to several colleagues and then left on vacation to South America. This email motivated a flurry of activity in research groups around the world. Lund, Fortnow, Karloff
showed that \#SAT is in IP (they added Nisan as a coauthor and the final paper is [LFK92]). Then Shamir showed that IP = PSPACE [Sha92] and Babai, Fortnow and Lund [BFL91] showed MIP = NEXPTIME. The entire story—as well as related developments—are described in Babai’s entertaining paper [Bab90].

Vadhan [Vad00] explores some questions related to the power of the prover.

The result that approximating the shortest vector is probably not NP-hard (as mentioned in the introduction) is due to Goldreich and Goldwasser [GG00].