Computational Complexity: A Modern Approach

Draft of a book: Dated January 2007

`Comments welcome!`

Sanjeev Arora and Boaz Barak
Princeton University
complexitybook@gmail.com

This is an Internet draft. Some chapters are more finished than others. References and attributions are very preliminary and we apologize in advance for any omissions (but hope you will nevertheless point them out to us).

Please send us bugs, typos, missing references or general comments to complexitybook@gmail.com — **Thank You!!**

Chapter 8

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

The standard notion of a mathematical proof follows the certificate definition of **NP**. That is, to prove that a statement is true one provides a sequence of symbols that can be written down in a book or on paper, and a valid sequence exists only for true statements. However, people often use more general ways to convince one another of the validity of statements: they *interact* with one another, with the person verifying the proof (henceforth the *verifier*) asking the person providing it (henceforth the *prover*) for a series of explanations before he is convinced.

It seems natural to try to understand the power of such interactive proofs from the complexity-theoretic perspective. For example, can one prove that a given formula is *not* satisfiable? (recall that is this problem is **coNP**-complete, it's not believed to have a polynomial-sized certificate). The surprising answer is *yes*. Indeed, interactive proofs turned out to have unexpected powers and applications. Beyond their philosophical appeal, interactive proofs led to fundamental insights in cryptographic protocols, the power of approximation algorithms, program checking, and the hardness of famous "elusive" problems (i.e., **NP**-problems not known to be in **P** nor to be **NP**-complete) such as *graph isomorphism* and *approximate shortest lattice vector.*

8.1 Warmup: Interactive proofs with a deterministic verifier

Let us consider what happens when we introduce *interaction* into the **NP** scenario. That is, we'd have an interrogation-style proof system where rather than the prover send a written proof to the verifier, the prover and verifier interact with the verifier asking questions and the prover responding, where at the end the verifier decides whether or not to accept the input. Of course, both verifier and prover can keep state during the interaction, or equivalently, the message a party sends at any point in the interaction can be a function of all messages sent and received so far. Formally, we make the following definition:

Definition 8.1 (Interaction of deterministic functions)
Let $f, g : \{0,1\}^* \to \{0,1\}^*$ be functions. A *k-round interaction* of f and g on input $x \in \{0,1\}^*$, denoted by $\langle f, g\rangle(x)$ is the sequence of the following strings $a_1, \ldots, a_k \in \{0,1\}^*$ defined as follows:

$$\begin{aligned} a_1 &= f(x) \\ a_2 &= g(x, a_1) \\ &\ldots \\ a_{2i+1} &= f(x, a_1, \ldots, a_{2i}) \\ a_{2i+2} &= g(x, a_1, \ldots, a_{2i+1}) \end{aligned} \tag{1}$$

(Where we consider a suitable encoding of i-tuples of strings to strings.)

The *output* of f (resp. g) at the end of the interaction denoted $\mathsf{out}_f\langle f, g\rangle(x)$ (resp. $\mathsf{out}_g\langle f, g\rangle(x)$) is defined to be $f(x, a_1, \ldots, a_k)$ (resp. $g(x, a_1, \ldots, a_k)$).

Definition 8.2 (Deterministic proof systems)
We say that a language L has a *k-round deterministic interactive proof system* if there's a deterministic TM V that on input $x, a_1, \ldots, a_i$ runs in time polynomial in $|x|$, satisfying:

$$\begin{aligned} \text{(Completeness)} x \in L \Rightarrow &\qquad \exists P : \{0,1\}^* \to \{0,1\}^* \, \mathsf{out}_V\langle V, P\rangle(x) = 1 \\ \text{(Soundness)} x \notin L \Rightarrow &\qquad \forall P : \{0,1\}^* \to \{0,1\}^* \, \mathsf{out}_V\langle V, P\rangle(x) = 1 \end{aligned}$$

The class **dIP** contains all languages with a $k(n)$-round deterministic interactive proof systems with $k(n)$ polynomial in n.

It turns out this actually does not change the class of languages we can prove:

Theorem 8.3
dIP = **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 $L \in$ **NP**. The certificate for membership is just the transcript $(a_1, a_2, \ldots, a_k)$ causing the verifier to accept. To verify this transcript, check that indeed $V(x) = a_1$, $V(x, a_1, a_2) = a_3$, …, and $V(x, a_1, \ldots, a_k) = 1$. If $x \in L$ then there indeed exists such a transcript. If there exists such a transcript $(a_1, \ldots, a_k)$ then we can define a prover function P to satisfy $P(x, a_1) = a_2$, $P(x, a_1, a_2, a_3) = a_4$, etc. We see that $\mathsf{out}_V\langle V, P\rangle(x) = 1$ and hence $x \in L$. ■

8.2 The class IP

In order to realize the full potential of interaction, we need to let the verifier be *probabilistic*. The idea is that, similar to probabilistic algorithms, the verifier will be allowed to come to a wrong conclusion (e.g., accept a proof for a wrong statement) with some small probability. However, as in the case of probabilistic algorithms, this probability is over the verifier's coins and the verifier will reject proofs for a wrong statement with good probability *regardless* of the strategy the prover uses. It turns out that the combination of interaction and randomization has a huge effect: as we will see, the set of languages which have interactive proof systems now jumps from **NP** to **PSPACE**.

EXAMPLE 8.4

As an example for a probabilistic interactive proof system, consider the following scenario: Marla claims to Arthur that she can distinguish between the taste of Coke (Coca-Cola) and Pepsi. To verify this statement, Marla and Arthur repeat the following experiment 50 times: Marla turns her back to Arthur, as he places Coke in one unmarked cup and Pepsi in another, choosing randomly whether Coke will be in the cup on the left or on the right. Then Marla tastes both cups and states which one contained which drinks. While, regardless of her tasting abilities, Marla can answer correctly with probability $\frac{1}{2}$ by a random guess, if she manages to answer correctly for *all* the 50 repetitions, Arthur can indeed be convinced that she can tell apart Pepsi and Coke.

To formally define this we extend the notion of interaction to *probabilistic* functions (actually, we only need to do so for the verifier). To model an interaction between f and g where f is probabilistic, we add an additional m-bit input r to the function f in (1), that is having $a_1 = f(x, r)$, $a_3 = f(x, r, a_1, a_2)$, etc. The interaction $\langle f, g\rangle(x)$ is now a random variable over $r \in_R \{0,1\}^m$. Similarly the output $\mathsf{out}_f\langle f, g\rangle(x)$ is also a random variable.

DEFINITION 8.5 (**IP**)

Let $k : \mathbb{N} \to \mathbb{N}$ be some function with $k(n)$ computable in poly(n) time. A language L is in $\mathbf{IP}[k]$ if there is a Turing machine V such that on inputs $x, r, a_1, \ldots, a_i$, V runs in time polynomial in $|x|$ and such that

$$\text{(Completeness)} \qquad x \in L \Rightarrow \exists P \Pr[\mathsf{out}_V\langle V, P\rangle(x) = 1] \geq 2/3 \qquad (2)$$

$$\text{(Soundness)} \qquad x \notin L \Rightarrow \forall P \Pr[\mathsf{out}_V\langle V, P\rangle(x) = 1] \leq 1/3. \qquad (3)$$

We define $\mathbf{IP} = \cup_{c\geq 1}\mathbf{IP}[n^c]$.

REMARK 8.6

The following observations on the class **IP** are left as an exercise (Exercise 1).

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.

Figure unavailable in pdf file.

Figure 8.1: Two isomorphic graphs.

2. Since the prover can use an arbitrary function, it can in principle use unbounded 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 the verifier's acceptance probability) using $\mathrm{poly}(|x|)$ space (and hence $2^{\mathrm{poly}(|x|)}$ time). Thus $\mathbf{IP} \subseteq \mathbf{PSPACE}$.

3. The probabilities of correctly classifying an input can be made arbitrarily close to 1 by using the same boosting technique we used for **BPP** (see Section **??**): to replace $2/3$ by $1-\exp(-m)$, sequentially repeat the protocol m times and take the majority answer. In fact, using a more complicated proof, it can be shown that we can decrease the probability without increasing the number of rounds using *parallel repetition* (i.e., the prover and verifier will run m executions of the protocol in parallel). We note that the proof is easier for the case of *public coin* proofs, which will be defined below.

4. Replacing the constant $2/3$ in the completeness requirement (2) 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 8.5.

5. In contrast replacing the constant $2/3$ by 1 in the soundness condition (3) is equivalent to having a deterministic verifier and hence reduces the class **IP** to **NP**.

6. We emphasize that the prover functions do not depend upon the verifier's random strings, but only on the messages/questions the verifier sends. 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 all the verifier's questions are simply obtained by tossing coins and revealing them to the prover (this is known as *public coins* or *Arthur-Merlin* proofs).

8.3 Proving that graphs are *not* isomorphic.

We'll now see an example of a language in **IP** that is not known to be in **NP**. 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 π of the labels of the nodes of G_1 such that $\pi(G_1) = G_2$. The graphs in figure **??**, 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 GI problem is the following: given two graphs G_1, G_2 (say in adjacency matrix representation) decide if they are isomorphic. Note that clearly $\mathsf{GI} \in \mathbf{NP}$, since a certificate is simply the description of the permutation π.

The graph isomorphism problem is important in a variety of fields and has a rich history (see [**?**]). Along with the factoring problem, it is the most famous **NP**-problem that is not known to be either in **P** or **NP**-complete. The results of this section show that GI is unlikely to be **NP**-complete, unless the polynomial hierarchy collapses. This will follow from the existence of the following proof system for the complement of GI: the problem GNI of deciding whether two given graphs are *not* isomorphic.

Protocol: Private-coin Graph Non-isomorphism

V**:** pick $i \in \{1, 2\}$ uniformly randomly. Randomly permute the vertices of G_i to get a new graph H. Send H to P.

P**:** identify which of G_1, G_2 was used to produce H. Let G_j be that graph. Send j to V.

V**:** accept if $i = j$; reject otherwise.

To see that Definition 8.5 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 in this case for every prover, $\Pr[V\text{accepts}] \leq 1/2$. This probability can be reduced to $1/3$ by sequential or parallel repetition.

8.4 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 8.7 (AM, MA)
For every k we denote by $\mathbf{AM}[k]$ the class of languages that can be decided by a k round interactive proof in which each verifier's message consists of sending a random string of polynomial length, and these messages comprise of all the coins tossed by the verifier. A proof of this form is called a *public coin* proof (it is sometimes also known an *Arthur Merlin* proof).[1]

We define by **AM** the class $\mathbf{AM}[2]$.[2] That is, **AM** is the class of languages with an interactive proof that consist of the verifier sending a random string, the prover responding with a message, and where the decision to accept is obtained by applying a deterministic polynomial-time function to the transcript. The class **MA** denotes the class of languages with 2-round public coins interactive proof with the prover sending the first message. That is, $L \in \mathbf{MA}$ if there's a proof system for L that consists of the prover first sending a message, and then the verifier tossing coins and applying a polynomial-time predicate to the input, the prover's message and the coins.

[1] 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. One "justification" for this model is that while Merlin cannot predict the coins that Arthur will toss in the future, Arthur has no way of hiding from Merlin's magic the results of the coins he tossed in the past.

[2] Note that $\mathbf{AM} = \mathbf{AM}[2]$ while $\mathbf{IP} = \mathbf{IP}[\text{poly}]$. While this is indeed somewhat inconsistent, this is the standard notation used in the literature. We note that some sources denote the class $\mathbf{AM}[3]$ by **AMA**, the class $\mathbf{AM}[4]$ by **AMAM** etc.

Note that clearly for every k, $\mathbf{AM}[k] \subseteq \mathbf{IP}[k]$. The interactive proof for GNI 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. Thus it may seem that allowing the verifier to keep its coins private adds significant power to interactive proofs, and so the following result should be quite surprising:

Theorem 8.8 ([**?**])
For every $k : \mathbb{N} \rightarrow \mathbb{N}$ with $k(n)$ computable in poly(n),

$$\mathbf{IP}[k] \subseteq \mathbf{AM}[k+2]$$

The central idea of the proof of Theorem 8.8 can be gleaned from the proof for the special case of GNI.

Theorem 8.9
GNI $\in \mathbf{AM}[k]$ *for some constant $k \geq 2$.*

The key idea in the proof of Theorem 8.9 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 $S = \{H : H \equiv G_1 \text{ or } H \equiv G_2\}$. Note that it is easy to prove that a graph H is a member of S, by providing the permutation mapping either G_1 or G_2 to H. The size of this set depends on whether G_1 is isomorphic to G_2. An n vertex graph G has at most $n!$ equivalent graphs. If G_1 and G_2 have each exactly $n!$ equivalent graphs (this will happen if for $i = 1, 2$ there's no non-identity permutation π such that $\pi(G_i) = G_i$) we'll have that

$$\text{if } G_1 \not\equiv G_2 \text{ then } |S| = 2n! \tag{4}$$
$$\text{if } G_1 \equiv G_2 \text{ then } |S| = n! \tag{5}$$

(To handle the general case that G_1 or G_2 may have less than $n!$ equivalent graphs, we actually change the definition of S to

$$S = \{(H, \pi) : H \equiv G_1 \text{ or } H \equiv G_2 \text{ and } \pi \in aut(H)\}$$

where $\pi \in aut(H)$ if $\pi(H) = H$. It is clearly easy to prove membership in the set S and it can be verified that S satisfies (4) and (5).)

Thus to convince the verifier that $G_1 \not\equiv G_2$, the prover has to convince the verifier that case (4) holds rather than (5). This is done by using a *set lower bound protocol*.

8.4.1 Set Lower Bound Protocol.

In a *set lower bound protocol*, the prover proves to the verifier that a given set S (where membership in S is efficiently verifiable) has cardinality at least K up to accuracy of, say, factor of 2. That is, if $|S| \geq K$ then the prover can cause the verifier to accept with high probability, while if $|S| \leq K/2$ then the verifier will reject with high probability, no matter what the prover does. By the observations above, such a protocol suffices to complete the proof of Theorem 8.9.

Tool: Pairwise independent hash functions.

The main tool we use for the set lower bound protocol is a *pairwise independent hash function collection*. This is a simple but incredibly useful tool that has found numerous applications in complexity theory and computer science at large (see Note 8.13).

DEFINITION 8.10 (PAIRWISE INDEPENDENT HASH FUNCTIONS)
Let $\mathcal{H}_{n,k}$ be a collection of functions from $\{0,1\}^n$ to $\{0,1\}^k$. We say that $\mathcal{H}_{n,k}$ is *pairwise independent* if for every $x, x' \in \{0,1\}^n$ with $x \neq x'$ and for every $y, y' \in \{0,1\}^k$, $\Pr_{h \in_R \mathcal{H}_{n,k}}[h(x) = y \ \wedge \ h(x') = y'] = 2^{-2n}$

Note that an equivalent formulation is that for every two distinct strings $x, x' \in \{0,1\}^n$ the random variable $\langle h(x), h(x') \rangle$ for h chosen at random from $\mathcal{H}_{n,k}$ is distributed according to the uniform distribution on $\{0,1\}^k \times \{0,1\}^k$.

Recall that we can identify the elements of $\{0,1\}^n$ with the *finite field* (see Section A.4 in the appendix), denoted $\mathrm{GF}(2^n)$, containing 2^n elements, whose addition $(+)$ and multiplication $(\cdot)$ operations satisfy the usual commutative and distributive laws, where and every element x has an additive inverse (denoted by $-x$) and, if nonzero, a multiplicative inverse (denoted by x^{-1}). The following theorem provides a construction of an *efficiently computable* pairwise independent hash functions (see also Exercise 4 for a different construction):

THEOREM 8.11 (EFFICIENT PAIRWISE INDEPENDENT HASH FUNCTIONS)
For every n define the collection $\mathcal{H}_{n,n}$ to be $\{h_{a,b}\}_{a,b \in \mathrm{GF}(2^n)}$ where for every $a, b \in \mathrm{GF}(2^n)$, the function $h_{a,b} : \mathrm{GF}(2^n) \rightarrow \mathrm{GF}(2^n)$ maps x to $ax + b$. Then, $\mathcal{H}_{n,n}$ is a collection of pairwise independent hash functions.

REMARK 8.12
Theorem 8.11 implies the existence of an efficiently computable pairwise independent hash functions $\mathcal{H}_{n,k}$ for every n, k: if $k > n$ we can use the collection $\mathcal{H}_{k,k}$ and reduce the size of the input to n by padding it with zeros. If $k < n$ then we can use the collection $\mathcal{H}_{n,n}$ and truncate the last $n - k$ bits of the output.

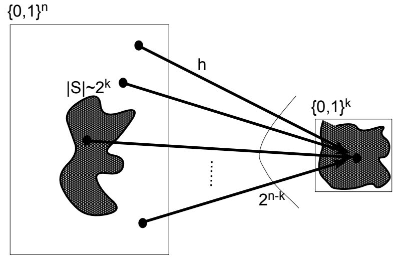
PROOF: For every $x \neq x' \in \mathrm{GF}(2^n)$ and $y, y' \in \mathrm{GF}(2^n)$, $h_{a,b}(x) = y$ and $h_{a,b}(x') = y'$ iff a, b satisfy the equations:

$$\begin{aligned} a \cdot x + b =& y \\ a \cdot x' + b =& y' \end{aligned}$$

These imply $a \cdot (x - x') = y - y'$ or $a = (y - y')(x - x')^{-1}$. Since $b = y - a \cdot x$, the pair $\langle a, b \rangle$ is completely determined by these equations, and so the probability that this happens over the choice of a, b is exactly one over the number of possible pairs, which indeed equals $\frac{1}{2^{2n}}$. ■

Note 8.13 (The Hashing paradigm)

A *hash function collection* is a collection of functions mapping a large universe, say $\{0,1\}^n$, to a smaller universe, say $\{0,1\}^k$ for $k \ll n$. Typically, we require of such a collection that it maps its input in a fairly uniform way to the output range. For example, if S is a subset of $\{0,1\}^n$ then we wish that, if h is chosen at random from the collection, then most elements of $\{0,1\}^k$ have roughly $|S|2^{-k}$ preimages in S (which is the expected number if h was a completely random function). In particular, if S has size roughly 2^k then we expect the mapping to be one-to-one or almost one-to-one, and so there should be a relatively small number of *collisions*: pairs $x \neq x' \in S$ such that $h(x) = h(x')$. Therefore, the image of S under h should look like this:



In databases, hash functions are used to maintain very efficient databases (that allow fast membership queries to a subset $S \subseteq \{0,1\}^n$ of size 2^k requiring only 2^k as opposed to 2^n bits of storage). In theoretical computer science, hash functions have a variety of uses. An example is Lemma 9.16 of the next chapter that shows that if the collection is pairwise independent and $S \subseteq \{0,1\}^n$ has size roughly 2^k, then with good probability the value 0^k will have exactly one preimage in S.

In all these cases it is important that the hash function is chosen at random from some collection independently of the choice of set S. It is easy to see that if k is small enough (e.g., $k < n/2$) then for *every* $h : \{0,1\}^n \to \{0,1\}^k$ there is a set $S \subseteq \{0,1\}^n$ of size 2^k that is "very bad" for h in the sense that all the members of S map to the same element under h.

Pairwise independent hash functions are but one example of a hash function collection. Several types of such collections are known in the literature featuring various tradeoffs between efficiency and uniformity of output.

The lower-bound protocol.

The lower-bound protocol is as follows:

Protocol: Goldwasser-Sipser Set Lowerbound

Conditions: $S \subseteq \{0,1\}^m$ is a set such that membership in S can be certified. Both parties know a number K. The prover's goal is to convince the verifier that $|S| \geq K$ and the verifier should reject if $|S| \leq \frac{K}{2}$. Let k be a number such that $2^{k-2} \leq K \leq 2^{k-1}$.

V: Randomly pick a function $h : \{0,1\}^m \to \{0,1\}^k$ from a pairwise independent hash function collection $\mathcal{H}_{m,k}$. Pick $y \in_R \{0,1\}^k$. Send h, y to prover.

P: Try to find an $x \in S$ such that $h(x) = y$. Send such an x to V, together with a certificate that $x \in S$.

V's output: If certificate validates that $x \in S$ and $h(x) = y$, accept; otherwise reject.

Let $p = \frac{K}{2^k}$. If $|S| \leq \frac{K}{2}$ then clearly $|h(S)| \leq \frac{p2^k}{2}$ and so the verifier will accept with probability at most $\frac{p}{2}$. The main challenge is to show that if $|S| \geq K$ then the verifier will accept with probability noticeably larger than $p/2$ (the gap between the probabilities can then be amplified using repetition). That is, it suffices to prove

Claim 8.13.1
Let $S \subseteq \{0,1\}^m$ satisfy $|S| \leq \frac{2^k}{2}$. Then,

$$\Pr_{h \in_R \mathcal{H}_{m,k}, y \in_R \{0,1\}^k}[\exists_{x \in S} h(x) = y] \geq \tfrac{3}{4}\tfrac{|S|}{2^k}\,.$$

Proof: For every $y \in \{0,1\}^m$, we'll prove the claim by showing that

$$\Pr_{h \in_R \mathcal{H}_{m,k}}[\exists_{x \in S} h(x) = y] \geq \tfrac{3}{4} p\,,$$

(where $p = |S|/2^k$). Indeed, for every $x \in S$ define the event E_x to hold if $h(x) = y$. Then, $\Pr[\exists_{x \in S} h(x) = y] = \Pr[\cup_{x \in S} E_x]$ but by the inclusion-exclusion principle this is at least

$$\sum_{x \in S} \Pr[E_x] - \tfrac{1}{2} \sum_{x \neq x' \in \S} \Pr[E_x \cap E'_x]$$

However, by pairwise independence, if $x \neq x'$, then $\Pr[E_x] = 2^{-k}$ and $\Pr[E_x \cap E'_x] = 2^{-2k}$ and so this probability is at least

$$\frac{|S|}{2^k} - \frac{1}{2}\frac{|S|^2}{2^k} = \frac{|S|}{2^k}\left(1 - \frac{|S|}{2^{k+1}}\right) \geq \frac{3}{4}p$$

■

Figure unavailable in pdf file.

Figure 8.2: **AM**$[k]$ looks like $\prod_k^p$, with the $\forall$ quantifier replaced by probabilitic choice.

Proving Theorem 8.9. The public-coin interactive proof system for GNI consists of the verifier and prover running several iterations of the set lower bound protocol for the set S as defined above, where the verifier accepts iff the fraction of accepting iterations was at least $0.6p$ (note that both parties can compute p). Using the Chernoff bound (Theorem A.18) it can be easily seen that a constant number of iteration will suffices to ensure completeness probability at least $\frac{2}{3}$ and soundness error at most $\frac{1}{3}$. ■

Remark 8.14

How does this protocol relate to the private coin protocol of Section 8.3? The set S roughly corresponds to the set of possible messages sent by the verifier in the protocol, where the verifier's message is a random element in S. If the two graphs are isomorphic then the verifier's message completely hides its choice of a random $i \in_R \{1, 2\}$, while if they're not then it completely reveals it (at least to a prover that has unbounded computation time). Thus roughly speaking in the former case the mapping from the verifier's coins to the message is 2-to-1 while in the latter case it is 1-to-1, resulting in a set that is twice as large. Indeed we can view the prover in the public coin protocol as convincing the verifier that its probability of convincing the private coin verifier is large. While there are several additional intricacies to handle, this is the idea behind the generalization of this proof to show that $\mathbf{IP}[k] \subseteq \mathbf{AM}[k+2]$.

Remark 8.15

Note that, unlike the private coins protocol, the public coins protocol of Theorem 8.9 does not enjoy perfect completeness, since the set lowerbound protocol does not satisfy this property. However, we can construct a perfectly complete public-coins set lowerbound protocol (see Exercise 3), thus implying a perfectly complete public coins proof for GNI. Again, this can be generalized to show that any private-coins proof system (even one not satisfying perfect completeness) can be transformed into a perfectly complete public coins system with a similar number of rounds.

8.4.2 Some properties of IP and AM

We state the following properties of **IP** and **AM** without proof:

1. (Exercise 5) $\mathbf{AM}[2] = BP \cdot \mathbf{NP}$ where $BP \cdot \mathbf{NP}$ is the class in Definition **??**. In particular it follows that$\mathbf{AM}[2] \subseteq \mathbf{\Sigma}_3^p$.
2. (Exercise 4) For constants $k \geq 2$ we have $\mathbf{AM}[k] = \mathbf{AM}[2]$. This "collapse" is somewhat surprising because $\mathbf{AM}[k]$ at first glance seems similar to **PH** with the $\forall$ quantifiers changed to "probabilistic $\forall$" quantifiers, where *most* of the branches lead to acceptance. See Figure 8.2.
3. It is open whether there is any nice characterization of $\mathbf{AM}[\sigma(n)]$, where $\sigma(n)$ is a suitably slow growing function of n, such as $\log \log n$.

8.4.3 Can GI be NP-complete?

We now prove that if GI is **NP**-complete then the polynomial hierarchy collapses.

Theorem 8.16 ([**?**])
If GI *is* **NP***-complete then* $\mathbf{\Sigma_2} = \mathbf{\Pi_2}$.

Proof: If GI is **NP**-complete then GNI is **coNP**-complete which implies that there exists a function f such that for every n variable formula φ, $\forall_y \varphi(y)$ holds iff $f(\varphi) \in \mathsf{GNI}$. Let

$$\psi = \exists_{x\in\{0,1\}^n} \forall_{y\in\{0,1\}^n} \varphi(x,y)$$

be a Σ_2SAT formula. We have that ψ is equivalent to

$$\exists_{x\in\{0,1\}^n} g(x) \in \mathsf{GNI}$$

where $g(x) = f(\varphi_{\restriction x})$.

Using Remark 8.15 and the comments of Section 8.4.2, we have that GNI has a two round **AM** proof with perfect completeness and (after appropriate amplification) soundness error less than 2^{-n}. Let V be the verifier algorithm for this proof system, and denote by m the length of the verifier's random tape and by m' the length of the prover's message and . We claim that ψ is equivalent to

$$\psi^* = \forall_{r\in\{0,1\}^{m'}} \exists_{x\in\{0,1\}^n} \exists_{a\in\{0,1\}^m} V(g(x), r, a) = 1$$

Indeed, by perfect completeness if ψ is satisfiable then ψ^* is satisfiable. If ψ is not satisfiable then by the fact that the soundness error is at most 2^{-n}, we have that there exists a single string $r \in \{0,1\}^m$ such that for every x with $g(x) \notin \mathsf{GNI}$, there's no a such that $V(g(x), r, a) = 1$, and so ψ^* is not satisfiable. Since ψ^* can easily be reduced to a Π_2SAT formula, we get that $\mathbf{\Sigma_2} \subseteq \mathbf{\Pi_2}$, implying (since $\mathbf{\Sigma_2} = \mathbf{co\Pi_2}$) that $\mathbf{\Sigma_2} = \mathbf{\Pi_2}$. ■

8.5 IP = PSPACE

In this section we show a surprising characterization of the set of languages that have interactive proofs.

Theorem 8.17 (LFKN, Shamir, 1990)
IP = **PSPACE**.

Note that this is indeed quite surprising: we already saw that interaction alone does not increase the languages we can prove beyond **NP**, and we tend to think of randomization as not adding significant power to computation (e.g., we'll see in Chapter 16 that under reasonable conjectures, **BPP** = **P**). As noted in Section 8.4.2, we even know that languages with constant round interactive proofs have a two round public coins proof, and are in particular contained in the polynomial hierarchy, which is believed to be a proper subset of **PSPACE**. Nonetheless, it turns out that the combination of sufficient interaction and randomness is quite powerful.

By our earlier Remark 8.6 we need only show the direction $\mathbf{PSPACE} \subseteq \mathbf{IP}$. To do so, we'll show that $\mathsf{TQBF} \in \mathbf{IP}[poly(n)]$. This is sufficient because every $L \in \mathbf{PSPACE}$ is polytime reducible to TQBF. We note that our protocol for TQBF will use public coins and also has the property that if 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 $\overline{\mathsf{3SAT}}$. 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 $\#\mathsf{SAT}$.) The idea of *arithmetization* introduced in this proof will also prove useful in our protocol for TQBF.

8.5.1 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 some finite field $\mathbb{F}$. Thus we have the following correspondence between formulas and polynomials when the variables take $0/1$ values:

$$\begin{aligned}
x \wedge y &\quad\longleftrightarrow\quad X \cdot Y \\
\neg x &\quad\longleftrightarrow\quad 1 - X \\
x \vee y &\quad\longleftrightarrow\quad 1 - (1 - X)(1 - Y) \\
x \vee y \vee \neg z &\quad\longleftrightarrow\quad 1 - (1 - X)(1 - Y)Z
\end{aligned}$$

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 $3m$ 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: we represent such a polynomial as a multiplication of all the degree 3 polynomials without "opening up" the parenthesis, and so $P_\varphi(X_1, X_2, \ldots, X_n)$ has a representation of size $O(m)$.) This conversion of φ to P_φ 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 $\mathbb{F}$ instead of just $0, 1$. As we will see, this gives the verifier unexpected power over the prover.

8.5.2 Interactive protocol for $\#\mathsf{SAT}_\mathsf{D}$

To design a protocol for $\overline{\mathsf{3SAT}}$ we give a protocol for $\#\mathsf{SAT}_\mathsf{D}$, which is a decision version of the counting problem $\#\mathsf{SAT}$ we saw in Chapter **??**:

$$\#\mathsf{SAT}_\mathsf{D} = \{\langle \phi, K \rangle : K \text{ is the number of satisfying assignments of } \phi\} \,.$$

and ϕ is a 3CNF formula of n variables and m clauses.

Theorem 8.18
$\#\mathsf{SAT}_\mathsf{D} \in \mathbf{IP}$.

PROOF: Given input $\langle \phi, K \rangle$, we construct, by arithmetization, P_ϕ. The number of satisfying assignments $\#\phi$ of ϕ is:

$$\#\phi = \sum_{b_1\in\{0,1\}} \sum_{b_2\in\{0,1\}} \cdots \sum_{b_n\in\{0,1\}} P_\phi(b_1, \ldots, b_n) \tag{6}$$

To start, the prover sends to the verifier a prime p in the interval $(2^n, 2^{2n}]$. The verifier can check that p is prime using a probabilistic or deterministic primality testing algorithm. All computations described below are done in the field $\mathbb{F} = \mathbb{F}_p$ of numbers modulo p. Note that since the sum in (6) is between 0 and 2^n, this equation is true over the integers iff it is true modulo p. Thus, from now on we consider (6) as an equation in the field $\mathbb{F}_p$. We'll prove the theorem by showing a general protocol, *Sumcheck*, for verifying equations such as (6).

Sumcheck protocol.

Given a degree d polynomial $g(X_1, \ldots, X_n)$, an integer K, and a prime p, we present an interactive proof for the claim

$$K = \sum_{b_1\in\{0,1\}} \sum_{b_2\in\{0,1\}} \cdots \sum_{b_n\in\{0,1\}} g(X_1, \ldots, X_n) \tag{7}$$

(where all computations are modulo p). To execute the protocol V will need to be able to evaluate the polynomial g for any setting of values to the variables. Note that this clearly holds in the case $g = P_\phi$.

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 in the variable X_1. 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 (7) is true, then we have $h(0) + h(1) = K$.

Consider the following protocol:

Protocol: Sumcheck protocol to check claim (7)

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 $s(X_1)$ (if the prover is not "cheating" then we'll have $s(X_1) = h(X_1)$).

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 (7) is true, the prover that always returns the correct polynomial will always convince V. If (7) is false then we prove that V rejects with high probability:

$$\Pr[V \text{ rejects } \langle K, g\rangle] \geq \left(1 - \frac{d}{p}\right)^n. \tag{8}$$

With our choice of p, the right hand side is about $1 - dn/p$, which is very close to 1 since $d \leq n^3$ and $p \gg n^4$.

Assume that (7) is false. We prove (8) by induction on n. For $n = 1$, 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 $h(X_1)$. Since the degree d nonzero polynomial $s(X_1) - h(X_1)$ has at most d roots, there are at most d values a such that $s(a) = h(a)$. Thus when V picks a random a,

$$\Pr_a[s(a) \neq h(a)] \geq 1 - \frac{d}{p}. \tag{9}$$

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 $\geq \left(1 - \frac{d}{p}\right)^{n-1}$. Thus we have

$$\Pr[V \text{ rejects}] \geq \left(1 - \frac{d}{p}\right) \cdot \left(1 - \frac{d}{p}\right)^{n-1} = \left(1 - \frac{d}{p}\right)^n \tag{10}$$

This finishes the induction.

■

8.5.3 Protocol for TQBF: proof of Theorem 8.17

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_ϕ. We have that $\Psi \in \mathsf{TQBF}$ if and only if

$$0 < \sum_{b_1 \in \{0,1\}} \prod_{b_2 \in \{0,1\}} \sum_{b_3 \in \{0,1\}} \cdots \prod_{b_n \in \{0,1\}} P_\phi(b_1, \ldots, b_n) \tag{11}$$

A first thought is that we could use the same protocol as in the $\#\mathsf{SAT_D}$ 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 look more closely at the polynomials that are are transmitted and their relation to the original formula. We'll change Ψ into a logically equivalent formula whose arithmetization does not cause the degrees of the polynomials to be so large. The idea is similar to the way circuits are reduced to formulas in the Cook-Levin theorem: we'll add auxiliary variables. Specifically, we'll change ψ to an equivalent formula ψ' that is not in prenex form in the following way: work from right to left and whenever encountering a $\forall$ quantifier on a variable x_i — that is, when considering a postfix of the form $\forall_{x_i}\tau(x_1,\ldots,x_i)$, where τ may contain quantifiers over additional variables $x_{i+1},\ldots,x_n$ — ensure that the variables $x_1,\ldots,x_i$ never appear to the right of another $\forall$ quantifier in τ by changing the postfix to $\forall x_i \exists x'_1,\ldots,x'_i (x'_1 = x_1)\wedge\cdots\wedge(x'_i = x_i)\wedge\tau(x_1,\ldots,x_n)$. Continuing this way we'll obtain the formula ψ' which will have $O(n^2)$ variables and will be at most $O(n^2)$ larger than ψ. It can be seen that the natural arithmetization for ψ' will lead to the polynomials transmitted in the sumcheck protocol never having degree more than 2.

Note that the prover needs to prove that the arithmetization of Ψ' leads to a number K different than 0, but because of the multiplications this number can be as large as 2^{2^n}. Nevertheless the prover can find a prime p between 0 and 2^n such that $K \mod p \neq 0$ (in fact as we saw in Chapter 7 a random prime will do). This finishes the proof of Theorem 8.17. ■

Remark 8.19

An alternative way to obtain the same result (or, more accurately, an alternative way to describe the same protocol) is to notice that for $x \in \{0,1\}$, $x^k = x$ for all $k \geq 1$. Thus, in principle we can convert any polynomial $p(x_1,\ldots,x_n)$ into a *multilinear* polynomial $q(x_1,\ldots,x_n)$ (i.e., the degree of $q(\cdot)$ in any variable x_i is at most one) that agrees with $p(\cdot)$ on all $x_1,\ldots,x_n \in \{0,1\}$. Specifically, for any polynomial $p(\cdot)$ let $L_i(p)$ be the polynomial defined as follows

$$L_i(p)(x_1,\ldots,x_n) = x_i P(x_1,\ldots,x_{i-1},1,x_{i+1},\ldots,x_n) + \\ (1-x_i)P(x_1,\ldots,x_{i-1},0,x_{i+1},\ldots,x_n) \quad (12)$$

then $L_1(L_2(\cdots(L_n(p)\cdots)$ is such a multilinear polynomial agreeing with $p(\cdot)$ on all values in $\{0,1\}$. We can thus use $O(n^2)$ invocations operator to convert (11) into an equivalent form where all the intermediate polynomials sent in the sumcheck protocol are multilinear. We'll use this equivalent form to run the sumcheck protocol, where in addition to having round for a $\sum$ or $\prod$ operator, we'll also have a round for each application of the operator L (in such rounds the prover will send a polynomial of degree at most 2).

8.6 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 8.9 requires the prover to produce, for some randomly chosen hash function h and a random element y in the range of h, a graph H such that $h(H)$ is isomorphic to either G_1 or G_2 and $h(x) = y$. This seems harder than just solving graph non-isomorphism.

2. The interactive proof for $\overline{\mathsf{3SAT}}$, a language in **coNP**, requires the prover to do #**P** computations, doing summations of exponentially many terms. (Recall that all of **PH** is in $\mathbf{P}^{\#\mathbf{P}}$.)

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 **??**, except the conclusion is stronger since **MA** is contained in $\mathbf{\Sigma_2}$ (indeed, a perfectly complete **MA**-proof system for L trivially implies that $L \in \mathbf{\Sigma_2}$).

Theorem 8.20
If $\mathbf{PSPACE} \subseteq \mathbf{P}/poly$ *then* $\mathbf{PSPACE} = \mathbf{MA}$.

Proof: If $\mathbf{PSPACE} \subseteq \mathbf{P}/poly$ 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 **??** below.

8.7 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 C for a program P is itself another program that may run P as a subroutine. Whenever P is run on an input x, C's job is to detect if P's answer is incorrect ("buggy") on that particular instance x. 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, since the fact that the checker did not detect a bug does not mean that P is a correct program in general, but only that P's answer on x is correct.

Definition 8.21
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 in the case that P is correct on x (i.e., $P(x) = C(x)$) but the program P is not correct everywhere, there is no guarantee on the output of the checker.

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 8.22 (Checker for Graph Non-Isomorphism)
The input for the problem of Graph Non-Isomorphism is a pair of labelled graphs $\langle G_1, G_2 \rangle$, 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 actually obtain the permutation π (). 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 $\langle G_1, H \rangle; \langle G_2, H \rangle$ 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 8.23
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 $P[C \text{ outputs "correct" }] \leq 2^{-k}$. Moreover, C runs in polynomial time.

8.7.1 Languages that have checkers

Whenever a language L has an interactive proof system where the prover can be implemented using oracle access to L, this implies that L has a checker. Thus, the following theorem is a direct consequence of the interactive proofs we have seen:

Theorem 8.24
The problems Graph Isomorphism (GI), Permanent (perm) and True Quantified Boolean Formulae (TQBF) have checkers.

Using the fact that **P**-complete languages are reducible to each other via **NC**-reductions, 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 8.25
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 additional evidence that checking is easier than actual computation.

8.8 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 8.5. 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** $\subseteq$ **MIP** since we can always simply ignore one prover. However,it turns out that **MIP** is probably strictly larger than **IP** (unless **PSPACE** = **NEXP**). That is, we have:

THEOREM 8.26 ([**?**])
NEXP = **MIP**

We will outline a proof of this theorem in Chapter **??**. One thing that we can do using two rounds is to force *non-adaptivity*. That is, consider the interactive proof as an "interrogation" where the verifier asks questions and gets back answers from the prover. If the verifier wants to ensure that the answer of a prover to the question q is a function only of q and does not depend on the previous questions the prover heard, the prover can ask the second prover the question q and accept only if both answers agree with one another. This technique was used to show that multi-prover interactive proofs can be used to implement (and in fact are equivalent to) a model of a "probabilistically checkable proof in the sky". In this model we go back to an **NP**-like notion of a proof as a static string, but this string may be huge and so is best thought of as a huge table, consisting of the prover's answers to all the possible verifier's questions. The verifier checks the proof by looking at only a few entries in this table, that are chosen randomly from some distribution. If we let the class $\mathbf{PCP}[r,q]$ be the set of languages that can be proven using a table of size 2^r and q queries to this table then Theorem 8.26 can be restated as

THEOREM 8.27 (THEOREM 8.26, RESTATED)
$\mathbf{NEXP} = \mathbf{PCP}[poly, poly] = \cup_c \mathbf{PCP}[n^c, n^c]$

It turns out Theorem 8.26 can be scaled down to to obtain $\mathbf{NP} = \mathbf{PCP}[\text{polylog}, \text{polylog}]$. In fact (with a lot of work) the following is known:

Theorem 8.28 (The **PCP** theorem, [?, ?])
$\mathbf{NP} = \mathbf{PCP}[O(\log n), O(1)]$

This theorem, which will be proven in Chapter 18, has had many applications in complexity, and in particular establishing that for many **NP**-complete optimization problems, obtaining an *approximately optimal* solution is as hard as coming up with the optimal solution itself. Thus, it seems that complexity theory has gone a full circle with interactive proofs: by adding interaction, randomization, and multiple provers, and getting to classes as high as **NEXP**, we have gained new and fundamental insights on the class **NP** the represents static deterministic proofs (or equivalently, efficiently verifiable search problems).

What have we learned?

- An *interactive proof* is a generalization of mathematical proofs in which the prover and polynomial-time probabilistic verifier interact.
- Allowing randomization and interaction seems to add significantly more power to proof system: the class **IP** of languages provable by a polynomial-time interactive proofs is equal to **PSPACE**.
- All languages provable by a *constant round* proof system are in the class **AM**: that is, they have a proof system consisting of the the verifier sending a single random string to the prover, and the prover responding with a single message.

Chapter notes and history

Interactive proofs were defined in 1985 by Goldwasser, Micali, Rackoff [?] for cryptographic applications and (independently, and using the public coin definition) by Babai and Moran [?]. The private coins interactive proof for graph non-isomorphism was given by Goldreich, Micali and Wigderson [?]. Simulations of private coins by public coins we given by Goldwasser and Sipser [?]. The general feeling at the time was that interactive proofs are only a "slight" extension of **NP** and that not even $\overline{\mathsf{3SAT}}$ has interactive proofs. The result **IP** = **PSPACE** was a big surprise, and the story of its discovery is very interesting.

In the late 1980s, Blum and Kannan [?] introduced the notion of program checking. Around the same time, manuscripts of Beaver and Feigenbaum [?] and Lipton [?] appeared. Inspired by some of these developments, Nisan proved in December 1989 that #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 [?]). Then Shamir showed that **IP** =**PSPACE** [?] and Babai, Fortnow and Lund [?] showed **MIP** = **NEXP**. The entire story —as well as related developments—are described in Babai's entertaining survey [?].

Vadhan [?] 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 [**?**].

Exercises

§1 Prove the assertions in Remark 8.6. That is, prove:

(a) Let $\mathbf{IP}'$ denote the class obtained by allowing the prover to be probabilistic in Definition 8.5. That is, the prover's strategy can be chosen at random from some distribution on functions. Prove that $\mathbf{IP}' = \mathbf{IP}$.

(b) Prove that $\mathbf{IP} \subseteq \mathbf{PSPACE}$.

(c) Let $\mathbf{IP}'$ denote the class obtained by changing the constant 2/3 in (2) and (3) to $1-2^{-|x|}$. Prove that $\mathbf{IP}' = \mathbf{IP}$.

(d) Let $\mathbf{IP}'$ denote the class obtained by changing the constant 2/3 in (2) to 1. Prove that $\mathbf{IP}' = \mathbf{IP}$.

(e) Let $\mathbf{IP}'$ denote the class obtained by changing the constant 2/3 in (3) to 1. Prove that $\mathbf{IP}' = \mathbf{NP}$.

§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 $\mathbf{IP}[2]$:

$$\text{QNR} = \{(y,m) : \ y \text{ is not a quadratic residue modulo } m\}.$$

§3 Prove that there exists a perfectly complete $\mathbf{AM}[O(1)]$ protocol for the proving a lowerbound on set size.

Hint: First note that in the current set lowerbound protocol we can have the prover choose the hash function. Consider the easier case of constructing a protocol to distinguish between the case $|S| \geq K$ and $|S| \leq \frac{1}{c}K$ where $c > 2$ can be even a function of K. If c is large enough the we can allow the prover to use *several* hash functions $h_1, \ldots, h_i$, and it can be proven that if i is large enough we'll have $\cup_i h_i(S) = \{0,1\}^k$. The gap can be increased by considering instead of S the set S^ℓ, that is the ℓ times Cartesian product of S.

§4 Prove that for every constant $k \geq 2$, $\mathbf{AM}[k+1] \subseteq \mathbf{AM}[k]$.

§5 Show that $\mathbf{AM}[2] = BP \cdot \mathbf{NP}$

§6 [**?**] Show that if $\mathbf{EXP} \subseteq \mathbf{P}/poly$ then $\mathbf{EXP} = \mathbf{MA}$.

Hint: The interactive proof for TQBF requires a prover that is a **PSPACE** machine.

§7 Show that the problem GI is downward self reducible. That is, prove that given two graphs G_1,G_2 on n vertices and access to a subroutine P that solves the GI problem on graphs with up to $n-1$ vertices, we can decide whether or not G_1 and G_2 are isomorphic in polynomial time.

§8 Prove that in the case that G_1 and G_2 are isomorphic we can obtain the permutation π mapping G_1 to G_2 using the procedure of the above exercise. Use this to complete the proof in Example 8.22 and show that graph isomorphism 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 isomorphism.

§9 Define a language L to be *downward self reducible* there's a polynomial-time algorithm R that for any n and $x \in \{0,1\}^n$, $R^{L_{n-1}}(x) = L(x)$ where by L_k we denote an oracle that solves L on inputs of size at most k. Prove that if L is downward self reducible than $L \in$ **PSPACE**.

§10 Show that **MIP** $\subseteq$ **NEXP**.

§11 Show that if we redefine multiprover interactive proofs to allow, instead of two provers, as many as $m(n) = \mathrm{poly}(n)$ provers on inputs of size n, then the class **MIP** is unchanged.

Hint: Show how to simulate 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.

8.A Interactive proof for the Permanent

The *permanent* is defined as follows:

Definition 8.29
Let $A \in F^{n\times n}$ be a matrix over the field F. The permanent of A is:

$$\mathsf{perm}(A) = \sum_{\sigma\in S_n}\prod_{i=1}^{n} a_{i,\sigma(i)}$$

The problem of calculating the permanent is clearly in **PSPACE**. In Chapter 9 we will see that if the permanent can be computed in polynomial time then **P** = **NP**, and hence this problem likely does not have a polynomial-time algorithm.

Although the existence of an interactive proof for the Permanent follows from that for #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.)

We use the following observation:

$$f(x_1, x_2, ..., x_n) := \mathsf{perm}\begin{bmatrix} x_{1,1} & x_{1,2} & \dots & x_{1,n} \\ x_{2,1} & \ddots & \dots & x_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ x_{n,1} & x_{n,2} & \dots & x_{n,n} \end{bmatrix}$$

is a degree n polynomial since

$$f(x_1, x_2, \dots, x_n) = \sum_{\sigma \in S_n} \prod_{i=1}^{n} x_{i,\sigma(i)}.$$

We now show two properties of the permanent problem. The first is *random self reducibility*, earlier encountered in Section **??**:

Theorem 8.30 (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:

- $f(x) := \mathsf{perm}(B)$ is a degree n univariate polynomial.
- For any fixed $b \neq 0$, $B(b)$ is a random matrix, hence the probability that oracle computes $\mathsf{perm}(B(b))$ 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 $1 - \frac{n+1}{n} \approx \frac{2}{3}$ the oracle will compute the permanent correctly on all matrices.
Recall the fact (see Section **??** in Appendix A) 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 assumption 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 **??**

Another property of the permanent problem is downward self reducibility, encountered earlier in context of SAT:

$$\mathsf{perm}(A) = \sum_{i=1}^{n} a_{1i} \mathsf{perm}(A_{1,i}),$$

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 8.31

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:

$$\forall i \in [n] \ . \ D_A(i) = A_{1,i}$$

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: $\mathsf{perm}(D_A(x))$ is a degree $n(n-1)$ polynomial in x.

8.A.1 The protocol

We now show an interactive proof for the permanent (the decision problem is whether $\mathsf{perm}(A) = k$ for some value k):

- Round 1: Prover sends to verifier a polynomial $g(x)$ of degree $n(n-1)$, which is supposedly $\mathsf{perm}(D_A(x))$.

- 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 2×2 matrix.

- Round $2(n-1)$: Verifier is left with a 2×2 matrix and calculates the permanent of this matrix and decides appropriately.

Claim 8.32

The above protocol is indeed an interactive proof for perm.

Proof: If $\mathsf{perm}(A) = k$, 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 $\mathsf{perm}(A) \neq k$. If on the first round, the polynomial $g(x)$ sent is the correct polynomial $D_A(x)$, then:

$$k \neq \sum_{i=1}^{m} a_{1,i} g(i) = \mathsf{perm}(A)$$

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 $n(n-1)$ points. Hence, the probability that they would agree on the randomly chosen point b_1 is at most $\frac{n(n-1)}{|F|}$. The same considerations apply to all subsequent rounds if exist, and the overall probability that the verifier will not accepts is thus (assuming $|F| \geq 10n^3$ and sufficiently large n):

$$\begin{aligned} Pr &\geq \left(1 - \frac{n(n-1)}{|F|}\right) \cdot \left(1 - \frac{(n-1)(n-2)}{|F|}\right) \cdot ... \left(1 - \frac{3 \cdot 2}{|F|}\right) \\ &\geq \left(1 - \frac{n(n-1)}{|F|}\right)^{n-1} \geq \frac{1}{2} \end{aligned}$$

■