

Computational Complexity Theory

Osamu Watanabe

Dept. Mathematical Computing Sciences, Tokyo Institute of Technology
 watanabe@is.titech.ac.jp

Part II Some Advanced Topics

One important approach in computational complexity theory — *structural approach* — is to investigate the structure of defined complexity classes, thereby obtaining better understanding of the “complexity” that these classes represent. In the last half of this course, let us see some of the recent important achievements of this structural approach.

1. Polynomial-Time Hierarchy and Interactive Proof

Some of relativized complexity classes such as P^{NP} , NP^{NP} , etc., are also important for investigating some concrete problems. But classes like NP^{NP} are not so easy to understand, at least not so intuitive. One way to get better understanding is to give a different formalization to these classes. For this we extend our definition of NP.

We simply generalize the *NP-condition* used to define the class NP. For example, consider the following condition.

$$x \in L \iff \exists u : |u| \leq q_L(|x|), \forall v : |v| \leq q_L(|x|) [R_L(x, u, v)].$$

A new complexity class Σ_2^P is the class of problems defined in this way with some polynomial q_L and polynomial-time computable predicate R_L . Similarly, classes Σ_3^P , Σ_4^P , ..., and classes Π_2^P , Π_3^P , ..., are defined using the following generalized NP-conditions.

$$\begin{aligned} \Sigma_3^P : x \in L &\iff \exists_{q_L} u_1, \forall_{q_L} v_1, \exists_{q_L} u_2 [R_L(x, u_1, v_1, u_2)] \\ \Sigma_4^P : x \in L &\iff \exists_{q_L} u_1, \forall_{q_L} v_1, \exists_{q_L} u_2, \forall_{q_L} v_2 [R_L(x, u_1, v_1, u_2, v_2)] \\ &\vdots \\ \Pi_2^P : x \in L &\iff \forall_{q_L} v_1, \exists_{q_L} u_1 [R_L(x, v_1, u_1)] \\ \Pi_3^P : x \in L &\iff \forall_{q_L} v_1, \exists_{q_L} u_1, \forall_{q_L} v_2 [R_L(x, v_1, u_1, v_2)] \\ &\vdots \end{aligned}$$

Notations:

$$\exists_{q_L} v_i \Leftarrow \exists v_i \in \{0, 1\}^* : |v_i| = q_L(|x|), \quad \forall_{q_L} u_i \Leftarrow \forall u_i \in \{0, 1\}^* : |u_i| = q_L(|x|).$$

By changing R_L appropriately, we may assume that witnesses are of the same polynomial length.

Now using these classes, we can give the following characterization.

Theorem 1. (Wrathall '77 and Stockmeyer '77)

$$\text{NP}^{\text{NP}} = \Sigma_2^{\text{P}}, \text{co-NP}^{\text{NP}} = \Pi_2^{\text{P}}, \text{NP}^{\text{NP}^{\text{NP}}} = \Sigma_3^{\text{P}}, \text{co-NP}^{\text{NP}^{\text{NP}}} = \Pi_3^{\text{P}}, \dots$$

Proof by Idea. We show that $\text{NP}^{\text{NP}} \subseteq \Sigma_2^{\text{P}}$.

Consider any problem L in NP^{NP} . Then there exist some relativized polynomial-time nondeterministic program A solving L by using some NP problem X as an oracle. Consider the computation of A on any input x of length ℓ , and let us formulate this computation by the Σ_2^{P} type formula. For the simplicity, we assume here that A asks exactly two queries on each computation path; moreover, assume that both queries are of length ℓ_1 , where ℓ_1 is polynomially bounded by ℓ .

Each computation path is expressed by some binary string w of some fixed length p_1 , where again p_1 is polynomially bounded by ℓ . If A were an ordinary nondeterministic program, its computation should be determined by such a binary string w . But since it makes two queries, the computation is not “easily” determined without knowing the answers of the oracle X to these queries. In fact, even the second query string may not be polynomial-time computable without knowing the answer to the first query. So, let us “guess” these two queries and the answers of X to them; note that each answer is expressed by one bit. That is, we express the computation of $A(x)$ as follows.

A accepts x (that is, $x \in L$)

$$\begin{aligned} \iff \exists w \in \{0, 1\}^{p_1}, \exists y_1, y_2 \in \{0, 1\}^{\ell_1}, \exists b_1, b_2 \in \{0, 1\} \\ \left[\begin{array}{l} (1) \text{ [the first query of } A \text{ on the path } w \text{ is } y_1 \text{]} \\ \wedge (2) \text{ [the second query of } A \text{ on the path } w \text{ and with the answer } b_1 \text{ is } y_2 \text{]} \\ \wedge (3) \text{ [} A \text{ outputs } 1 \text{ on } w \text{ and with the assumed answers } b_1 \text{ and } b_2 \text{]} \\ \wedge (4) \text{ [the first query } y_1 \text{ gets 'yes' answer from } X \text{ iff } b_1 = 1 \text{]} \\ \wedge (5) \text{ [the second query } y_2 \text{ gets 'yes' answer from } X \text{ iff } b_2 = 1 \text{]} \end{array} \right] \end{aligned}$$

It is easy to see that (1) \sim (3) can be checked *deterministically* in polynomial time. On the other hand, this may not be the case for (4) and (5) because $X \in \text{NP}$. But, for example, (4) can be expressed as follows with some q_X and R_X .

$$\begin{aligned} (4) \iff [b_1 = 1 \wedge [y_1 \in X]] \vee [b_1 = 0 \wedge [y_1 \notin X]] \\ \iff [b_1 = 1 \wedge \exists_{q_X(\ell_1)} u_1 [R_X(y_1, u_1)]] \vee [b_1 = 0 \wedge \forall_{q_X(\ell_1)} v_1 [\neg R_X(y_1, v_1)]] \\ \iff \exists_{q_X(\ell_1)} u_1, \forall_{q_X(\ell_1)} v_1 [(4') [b_1 = 1 \wedge R_X(y_1, u_1)] \vee [b_1 = 0 \wedge \neg R_X(y_1, v_1)]] \end{aligned}$$

Hence, as a whole, we can restate “ A accepts x ” by the following Σ_2^{P} -type formula.

$$\exists_{p_1} w, \exists_{\ell_1} y_1, y_2, \exists_1 b_1, b_2, \exists_{q_X(\ell_1)} v_1, v_2, \forall_{q_X(\ell_1)} u_1, u_2 [(1) \sim (3) \wedge (4') \wedge (5')].$$

We may regard $\text{NP} = \Sigma_1^{\text{P}}$ and $\text{co-NP} = \Pi_1^{\text{P}}$. Also define $\Sigma_0^{\text{P}} = \Pi_0^{\text{P}} = \text{P}$. For relativized classes like P^{NP} , we introduce the class $\Delta_k^{\text{P}} = \text{P}^{\Sigma_{k-1}^{\text{P}}}$. Then the following relationship holds from the definition or as a corollary of Theorem ??.

Proposition 2.

$$\text{P} = \Sigma_0^{\text{P}} = \Pi_0^{\text{P}} \subseteq \text{NP} = \Sigma_1^{\text{P}}, \text{co-NP} = \Pi_1^{\text{P}} \subseteq \Sigma_2^{\text{P}}, \Pi_2^{\text{P}} \subseteq \Sigma_3^{\text{P}}, \Pi_3^{\text{P}} \subseteq \dots$$

Corollary 3. $\Sigma_{k-1}^{\text{P}} \cup \Pi_{k-1}^{\text{P}} \subseteq \Delta_k^{\text{P}} \subseteq \Sigma_k^{\text{P}} \cap \Pi_k^{\text{P}}$.

Polynomial-time hierarchy PH is defined to be the union of these Σ_k^{P} classes.

Notice that the order of quantifiers cannot be changed in general (unless they are of the same type). Thus, from our Σ_2^{P} formulation of NP^{NP} , it is clear that $\text{NP} \neq \text{NP}^{\text{NP}}$ unless $\text{NP} = \Pi_1^{\text{P}}$ (= co-NP). In fact, it is easy to see that $\text{NP} = \text{NP}^{\text{NP}}$ (and thus, the whole PH collapses NP) if and only if $\text{NP} = \text{co-NP}$.

Corollary 4. $\text{NP} = \text{co-NP} \iff \text{NP} = \text{PH}$. In general, $\Sigma_k^{\text{P}} = \Pi_k^{\text{P}} \iff \Sigma_k^{\text{P}} = \text{PH}$.

• **Different view: two person game and PSPACE**

The alternation of quantifiers can be viewed as a play of some two player game. For example, consider the following *quantified Boolean formula* Q .

$$Q = \exists_q u_1, \forall_q v_1, \exists_q v_2, \forall_q u_2 [\Phi(x, u_1, v_1, u_2, v_2)].$$

The existential quantifier is for (the choice of) you, and the universal quantifier is for (the choice of) the opponent. The goal of the game (from your point of view) is to make Φ true; on the other hand, your opponent's goal is to make Φ false. The number of alternations can be regarded as the number of turns (or the depth) of the game. (Here for the simplicity, we assume that every formula starts with an existential quantifier and ends with an universal quantifier.)

Let us call this game an *alternating satisfying game* (over a *polynomial-time* Boolean predicate). Note that each game is defined by a predicate Φ that, we assume, polynomial-time computable, and each play of the game is defined by a given input x . We say that an alternating satisfying game *solves* a problem L if $x \in L$ if and only if the \exists -player wins at the play for x . We can further generalize this two player alternating satisfying game so that the depth of each play is not constant but $d(|x|)$ on each input x for some given function. With this computation model, we can introduce the following new complexity class.

$$\text{ADepth}(d) = \left\{ L : \begin{array}{l} L \text{ is solvable by some alternating satisfying game} \\ \text{whose game depth is } d(|x|) \text{ on each input } x \end{array} \right\}.$$

Then we have $\text{PH} = \text{ADepth}(O(1))$.

Theorem 5. $\text{PSPACE} = \bigcup_{p:\text{poly}} \text{ADepth}(p)$.

Proof. Consider any problem L in PSPACE, and let A be a deterministic program solving L , whose space complexity is bounded by some polynomial p .

Consider the execution of A on any input of length ℓ . Since strings kept in A 's each register is at most of $p(\ell)$ bits, the state of the program at each step can be expressed by a binary string of length $q = cp(\ell)$ for some constant $c > 0$. In fact, with these states and direct edges connecting them, we can express the computation of A on *all* inputs of length ℓ as a graph $G_{A,\ell}$.

More specifically, the graph $G_{A,\ell}$ consists of vertices corresponding states, each of which is labeled by binary string in $\{0, 1\}^q$, and the graph has an edge from one vertex S_1 to another S_2 if and only if the execution of A moves from S_1 to S_2 in one step. Then for any input $x \in \{0, 1\}^\ell$, we have $x \in L$ if and only if there is a path from the *initial vertex* for x , the vertex corresponding to the initial state of $A(x)$, to the *accepting vertex*, a vertex corresponding to the halting state yielding 1 (i.e., 'yes'). Notice here that the length of such an *accepting path* is at most 2^q ; in fact, we may assume that the length is exactly $2^{d(\ell)}$ for some polynomial d .

Now with this graph, we define an alternating satisfying game for simulating A on x for any given $x \in \{0, 1\}^\ell$. Intuitively, you want to convince the opponent that there is a path from the initial vertex S_0 for x to the accepting vertex S_{acc} . For this, you point out the vertex S_1 that is located exactly at the middle of the path. Then the opponent asks you to show either (i) there is a path from S_0 to S_1 , or (ii) there is a path from S_1 to S_{acc} . In other words, the opponent's move is either 0 indicating (i) or 1 indicating (ii). Then depending on this move, you show for S_2 either (0) the vertex located at the middle of S_0 and S_1 , or (1) the vertex located at the middle of S_1 and S_{acc} . The play proceeds in this way for $d(\ell) - 1$ pairs of moves. Then after these moves (if your choices have been correct), there must be an edge between the current vertices S_i and S_j , i.e., A moves from S_i to S_j in one step. (The indices i and j are determined easily from the sequence of the opponent moves.)

Since you cannot predict this opponent's move in advance, you have to give the correct middle vertex for S_1 , and so on; otherwise, the opponent certainly finds the problematic part. In particular, if there is no accepting path for a given x , then you cannot win the game no matter how you choose moves. Therefore, we have the following relation, which proves that $L \in \text{ADepth}(d')$ for $d'(n) = 2d(n) - 2$.

$$x \in L \iff \exists S_1, \forall v_1 \in \{0, 1\}, \exists S_2, \forall v_2 \in \{0, 1\}, \dots \exists S_{d(\ell)-1}, \forall v_{d(\ell)-1} \in \{0, 1\} [S_i \Rightarrow_A S_j].$$

Remarks:

The class $\text{ADepth}(d)$ is introduced for the explanation here, and it is not commonly

used. Standard complexity classes related to this $\text{ADepth}(d)$ is alternating complexity classes that are defined by using *alternating computation*, which is a generalization of nondeterministic computation.

The above proof also shows that the following satisfiability problem — the Quantified Boolean Formula satisfiability problem — is PSPACE-complete.

$$\text{QBF} = \{ Q : Q \text{ is a } \textit{true} \text{ quantified Boolean formula with no free variable } \}.$$

• **Complexity analysis using classes in PH — one of my favorites ;–)**

While it seems hard to show (unconditional) nontrivial circuit lower bounds to problems in, say NP, we can indeed prove some interesting lower bounds for problems in PH.

The key step is the following theorem.

Theorem 6. (Karp and Lipton '80) If $\text{NP} \subseteq \text{P/poly}$, then $\text{PH} = \Sigma_2^{\text{P}} \cap \Pi_2^{\text{P}}$.

Theorem 7. (Kannan '82) For any polynomial p , there exists a problem in $\Sigma_2^{\text{P}} \cap \Pi_2^{\text{P}}$ that is not in $\text{SIZE}[p]$.

Remarks:

The collapsing consequence has been improved to the class ZPP^{NP} (Bshouty et al '96 and Köbler and Watanabe '98), and more recently, it has been improved further to the class S_2^{P} (Cai '01). Accordingly, the above lower bound can be shown for these lower classes.

Proof of Theorem ??.

Consider any polynomial p . First we define a problem D in Δ_3^{P} that is not in $\text{SIZE}[p]$. This is possible by noting that the following predicate is Σ_2^{P} -computable.

$$\textit{kill}(\bar{\ell}, u) \stackrel{\text{def}}{\iff} \exists_{p(\ell)+1} w \succ u, \forall C : \text{circuit of size } \leq p(\ell) [C \text{ does not solve } L_{\ell,w}].$$

Here by $w \succ u$, we mean that u is a prefix of w , and the problem $L_{\ell,w}$ is defined as follows. (Recall that $\bar{\ell} = 0^\ell$. Let $\textit{bin}(x)$ denote the number represented by x .)

$$L_{\ell,w} = \{ x \in \{0,1\}^\ell : \text{the } \textit{bin}(x)\text{th bit of } w \text{ is } 1 \}.$$

Next consider any NP-complete problem, for example, SAT. Suppose that SAT is in $\text{SIZE}[p]$. Then we are done because the class NP contains a problem (i.e., SAT) not in $\text{SIZE}[p]$. But on the other hand, if SAT is in $\text{SIZE}[p]$, then by Theorem ??, the PH collapses to $\Sigma_2^{\text{P}} \cap \Pi_2^{\text{P}}$; in particular, the above $D \notin \text{SIZE}[p]$ belongs to $\Sigma_2^{\text{P}} \cap \Pi_2^{\text{P}}$.

Remarks:

This proof has “nonconstructive” flavor. That is, it does not specify a single problem witnessing $\Sigma_2^P \cap \Pi_2^P - \text{SIZE}[p] \neq \emptyset$. I have been trying to define such a problem. But surprisingly, the problem seems very difficult.

For randomized classes, the following relations are important.

Theorem 8. (Sipser and Gacs '83, Lautemann '83, Zachos '86) $\text{BPP} \subseteq \Sigma_2^P \cap \Pi_2^P$.

Theorem 9. (Toda '91) $\text{PH} \subseteq \text{P}^{\text{PP}}$.

• Interactive proofs

The notion of “interactive proof” has been introduced in several contexts, which turns out to be an important computation model defining various interesting complexity classes.

The interactive proof generalizes our alternating satisfying game on Boolean predicates in two ways. One is to introduce “randomized” computation, and the other is to separate a “message” from a “choice” (or, “move”). Below let us discuss these generalizations separately.

First let us consider the latter generalization. An *alternating interactive proof* is a pair of two players, *prover* P and *verifier* V , communicating each other by sending messages in turn. (Each turn is called a *round*.) We may consider a prover is an oracle, an imaginary magic device with infinite computation power, and a verifier is a polynomial-time nondeterministic program, where “nondeterministic choice” or “move” is used only for generating messages. For any input x (that is accessible by both prover and verifier), the goal of the game is to convince the verifier that x is a positive instance (of a given problem L). We say that (P, V) is an *alternating interactive proof for* L if the following holds.

$$\forall x \in \{0, 1\}^* [x \in L \iff [P \text{ can convince } V \text{ on } x \text{ for every choice of } V]].$$

Define the following complexity class.

$$\text{AIP}(r) = \{ L : L \text{ has an alternating interactive proof of } r \text{ rounds} \}.$$

Remarks:

The notion of “alternating” interactive proof is not common, and the class $\text{AIP}(r)$ is defined only for the explanation in this lecture. A randomized version, which we will define next, is the standard one that is usually used.

Example 1. (AIP for the Graph NonIsomorphism (GNI) problem)

Consider the following Graph NonIsomorphism (GNI) problem.

$$\text{GNI} = \{ (G_1, G_2) : G_1 \text{ is not isomorphic to } G_2 \}.$$

What follows is an alternating interactive proof for GNI. The proof starts from a verifier's turn.

input A pair (G_1, G_2) of graphs of n vertices;
V: choose a permutation p of n vertices and $i \in \{1, 2\}$;
 $G \leftarrow p(G_i)$;
send G to the prover;
P: $j \leftarrow$ the index such that $G = p(G_j)$ holds;
send j to the verifier;
V: **if** $i = j$ **then** **halt**(1) **else** **halt**(0);

Clearly, the prover can always convince the verifier if a given (G_1, G_2) is a pair of nonisomorphic graph. On the other hand, if G_1 is isomorphic to G_2 , then the prover fails to convince the verifier for some choice of the verifier (in fact, for at least the half of its choices). Thus, (P, V) is an alternating interactive proof for GNI. Since the protocol requires 2 rounds, this proof shows that GNI is in AIP(2).

Intuitively, a prover corresponds to the \exists -player of the alternating satisfying game, while a verifier corresponds to the \forall -player. More precisely, we have, e.g., $\Sigma_k^P \subseteq \text{AIP}(2k)$. The important difference is that a message, though computed from a choice, can be different from the choice. In fact, it is the key point of the above proof that the chosen i is hidden to the prover. A choice like this is called a *private choice* while a guess in the alternating satisfying game is called a *public choice*.

The problem of AIP is that nondeterministic verifiers are too powerful even if it is acceptable to assume infinitely powerful provers. The next generalization idea is to use "randomized" verifiers.

The definition of a (randomized) *interactive proof* (P, V) is almost the same as the alternating one, except that verifier's choice is regarded as a randomly one, and that we require the following bounded error probability condition. (Here again ϵ can be any constant $< 1/2$.)

$$\begin{aligned} x \in L &\implies \Pr_V\{P \text{ can convince } V \text{ on } x\} > 1 - \epsilon, \\ x \notin L &\implies \Pr_V\{P \text{ can convince } V \text{ on } x\} < \epsilon. \end{aligned}$$

The class $\text{IP}(r)$ is defined as follows. In particular, IP is defined as $\cup_{p:\text{poly}} \text{IP}(p)$.

$$\text{IP}(r) = \{ L : L \text{ has an interactive proof of } r \text{ rounds} \}.$$

Recall that the error probability of the interactive proof for GNI defined in Example ?? is at most $1/2$; thus, running this proof twice, we can reduce the error probability at most $1/4$, and this modified interactive proof satisfies the above bounded error probability condition (with $\epsilon = 1/4 + \Delta$). Hence GNI is in $IP(4)$.

Remarks:

1. Surprisingly, the difference between “private” and “public” is not so essential in the randomized proof. One can show (Goldwasser and Sipser ’89) that every IP can be simulated by an interactive proof using only public random sequences, which is called *Arthur-Merlin game*.
2. What is more surprising is that every constant round IP can be simulated by some Arthur-Merlin game with *with two rounds* (The above result + Babai and Moran ’88). This implies that $IP(O(1)) \subseteq IP(2)$. That is, no hierarchy exists; this contrasts to our belief that PH consists of infinite Σ_k^P classes.
3. The above results show that the difference between “private” and “public” is not computationally important. This difference is, nevertheless, important for some other contexts; for example, in interactive proofs as security protocols.

The most important interactive proof may be the one for QBF problem, which proves the following relation.

Theorem 10. (Shamir ’92) $IP = PSPACE$.

Remarks:

1. $IP \subseteq PSPACE$ is clear. Shamir showed that QBF is in $IP(p)$ for some polynomial p , which implies $PSPACE \subseteq IP$.
2. The breakthrough is given by Lund, Fortnow, Karloff, and Nisan ’92, who proved that $P^{PP} \subseteq IP$. In this proof, they developed an important technique — *arithmetization* of a Boolean formula — which is also the key for proving the above theorem.
3. Once a Boolean formula is arithmetized, we can use (randomized) algebraic techniques for checking its satisfiability (with the help of some prover). This is also the key to get the recent PCP characterization of the class NP.

2. One-Way Function and Pseudo Random Sequence Generator

Investigating complexity of problems, or more precisely, hardness of problems, is an important mathematical subject, but it does not have any practical meaning. Most people including complexity theorists would have said so in the early 70's, but the situation has been completely changed since Deffie and Hellmann introduced the notion of “public-key cryptography” in 1976. We have realized that computational hardness can be used as a basis of security. Using the hardness some NP problems, many cryptographic tools and cryptosystems have been proposed, and the area studying this type of cryptography — *computational cryptography* — has become one branch of computational complexity theory. Here we survey some of the fundamental results in computational complexity theory developed for computational cryptography.

The “one-wayness” may be the most fundamental notion in computational cryptography. Roughly speaking, a function f is called “one-way” if f is easy to compute but its inverse f^{-1} is hard. This is the basic requirement for any cryptographic encoding function E , a function mapping a plain text into a cipher text, because the security of encoded cipher texts depends on the hardness of E^{-1} .

The one-wayness can be formulated in the following way.

Definition 1. A function f is called (polynomial-time) *one-way* if (i) it is polynomial-time computable, and (ii) for any polynomial-time program A , and for any polynomial q , the following inequality holds for any sufficiently large ℓ .

$$\Pr_{x \in \{0,1\}^\ell} \{ A(f(x)) = f^{-1}(f(x)) \} < 1/q(\ell).$$

Remarks:

1. In this section, we will consider problems of computing general (not only Boolean) functions.
2. For the sake of simplicity, we will assume that one-way functions (or candidates) are one-to-one, and we will sometimes consider only *permutations*, i.e., total, one-to-one, and length-preserving functions over $\{0,1\}^*$. That is, they map for any $x \in \{0,1\}^\ell$ to some $y \in \{0,1\}^\ell$ for each $\ell \geq 0$. For any one-to-one function f , we can regard f^{-1} as a function; in particular, if f is a permutation, so is f^{-1} .
3. The condition of Definition ?? is a slightly simplified version. In general, randomized programs are also allowed for A .

This one-wayness is closely related to the P vs. NP conjecture.

Theorem 11. For any polynomial-time computable permutation f , there exists some $X_f \in \text{NP}$ such that f^{-1} is polynomial-time computable relative to X_f . Hence, if $X_f \in \text{P}$, then f^{-1} is not one-way.

Remarks:

1. We will abuse notations for complexity classes to include functions. Then the above theorem claims that $f^{-1} \in P^{X_f}$.
2. From this theorem, we have $P = NP \implies$ no one way function exists.
3. What we want for is the converse relation: namely, $P \neq NP \implies$ some one way function exists. But there seems to be some gap; the one-wayness is based on the average-case hardness, whereas $P \neq NP$ is about the worst-case hardness. The worst-case hardness does not necessarily imply the average-case hardness.

Proof. For a given polynomial-time computable permutation f , we can define X_f as follows.

$$X_f = \{ (y, u) : \exists x \succ u [f(x) = y] \}.$$

Then it is easy to see that a standard binary search algorithm can compute $f^{-1}(y)$ with help of (the oracle solving) this X_f .

• **The degree of one-wayness and concrete examples**

Although showing concrete one-way functions is important, it is not so easy to think of any function satisfying this rather strong one-wayness requirement. On the other hand, there are some problems (e.g., the factorization problem) from which we can define a function with some sort of inverting hardness. For discussing such examples, we consider the following weaker one-wayness.

Definition 2. A function f is called *weak one-way* if (i) it is polynomial-time computable, and (ii) there exists a polynomial q_0 such that for any polynomial-time program A , the following inequality holds for any sufficiently large ℓ .

$$\Pr_{x \in \{0,1\}^\ell} \{ A(f(x)) \neq f^{-1}(f(x)) \} \geq 1/q_0(\ell).$$

Remarks:

1. When comparing these one-wayness notions, we refer the original one as *strong one-wayness*.
2. This hardness condition is still stronger than the worst-case polynomial-time non-computability.

Example 2. Consider the following function.

$$pmult((u, v)) = \begin{cases} u \times v, & \text{if } u < v \text{ and both } u \text{ and } v \text{ are prime numbers, and} \\ (u, v) & \text{otherwise.} \end{cases}$$

This function is weak one-way provided that the factorization of the product of two large prime numbers is “hard” on average.

Although this function is not a permutation, since it is one-to-one, we can modify it so that it becomes one-to-one and length regular (i.e., the output length is uniquely determined from the input length), which is enough for the most of our discussion.

Now the following theorem gives us a bridge from this concrete example to cryptographically secure one-way functions.

Theorem 12. (Yao 1982) From any weak one-way function, we can construct a strong one-way function.

• Pseudo random binary sequence generators

One of the important cryptographic applications of one-way functions is the construction of a secure pseudo random bit generator, or a secure pseudo binary sequence generator (prg in short).

For defining prg formally, we should first clarify the notion of “pseudo randomness”. For this, the following clever approach has been invented: First define the notion of “statistical test”, and then define “pseudo random sequence” as a sequence that passes all statistical tests.

Let us first define our notion of “statistical test”. In the following, a *generator* is a function g generating some binary sequence from a given binary sequence *seed*. We assume that all generators are length regular; that is, its output length is uniquely determined from the input length.

Definition 3. For any generator g , a polynomial-time randomized program T is called a *statistical test* for (the nonrandomness of) g if the following holds for some polynomial p_0 and for any sufficiently large n . (We use ℓ to denote the output length of g on inputs of length n .)

$$\left| \Pr_{T, s \in \{0,1\}^n} \{T(g(s)) = 1\} - \Pr_{T, r \in \{0,1\}^\ell} \{T(r) = 1\} \right| \geq \frac{1}{p_0(\ell)}.$$

On the contrary, if T fails to satisfy this condition, then we say that g passes the test T .

Now the notion of prg is defined as follows.

Definition 4. A generator g is called a *pseudo random binary sequence generator* (or, more simply, *pseudo random sequence generator*) if $|g(s)| > |s|$ for all seed s and g passes all statistical tests.

It is easy to see that every prg has the strong one-wayness property. But, on the other hand, the construction of prg from any one-way function is quite involved. We explain the construction step by step.

First we introduce somewhat weaker notion of statistical test.

Definition 5. For any generator g , a polynomial-time randomized program \mathbf{N} is called a *next bit test* for (the nonrandomness of) g if the following holds for some polynomial p_0 and for any sufficiently large n .

$$\exists i \in \{0, \dots, \ell - 1\} \left[\Pr_{\mathbf{N}, s \in \{0,1\}^n} \{ \mathbf{N}(\ell, g(s)_{\leftarrow i}) = g(s)_{i+1} \} \geq \frac{1}{2} + \frac{1}{p_0(\ell)} \right].$$

Here by $g(s)_{\leftarrow i}$ we mean the prefix of $g(s)$ up to the i th bit, and let $g(s)_{i+1}$ denote the $(i + 1)$ th bit of $g(s)$.

Theorem 13. (Yao '82) g passes all statistical tests $\iff g$ passes all next bit tests.

From this theorem, we can set our goal as defining g with $|g(s)| = |s| + 1$ that passes all next bit tests. Though this g 's expansion property is weak, it is certainly a prg.

For defining such g from any one-way function, the following notion is important.

Definition 6. For any function f , a predicate H is called a *hard-core predicate* for f if (i) it is polynomial-time computable, and (ii) for any polynomial-time program \mathbf{A} , and for any polynomial q , the following inequality holds for any sufficiently large ℓ .

$$\Pr_{x \in \{0,1\}^\ell} \{ \mathbf{A}(f(x)) = H(x) \} < \frac{1}{2} + \frac{1}{q(\ell)}.$$

Remarks:

The above definition makes sense for, at least, one-to-one functions. For one-to-one functions, the value of $H(x)$ should be determined from $f(x)$; what is required above is that it is just *hard* to compute from $f(x)$. On the other hand, one can easily define a *two-to-one* function f and a predicate H so that nothing can compute $H(x)$ from $f(x)$, simply because $H(x)$ is not determined from $f(x)$.

Theorem 14. (Goldreich and Levin '89) For any one-way function f , define h and H as follows. Then H is a hard core predicate for h .

$$\begin{aligned} h(w) &= f(x)z, \quad \text{where } x, z \in \{0,1\}^\ell, \text{ and} \\ H(w) &= x \cdot z = (x_1 \cdot z_1) \oplus (x_2 \cdot z_2) \oplus \dots \oplus (x_n \cdot z_n). \end{aligned}$$

Now we are ready to achieve our goal. In fact, the definition of g and its justification is easy.

Theorem 15. Let h be a permutation with a hard core predicate H . Then $g(w) = h(w)H(w)$ passes all next bit tests; hence, g is prg.

Remarks:

A similar construction is shown from any one-to-one and length regular one-way function (Håstad, Impagliazzo, Levin and Luby '99).

Although we have succeeded defining prg g , the obtained one is very weak and it can expand only one bit to the sequence that is given as a seed. But it is provable that from such a prg, for any polynomial e , we can define some prg that expands a given random seed $s \in \{0, 1\}^n$ by $e(n)$ bits.

• **Derandomization**

Pseudo random sequence generators are important also for our theoretical investigation of complexity classes. In particular, for studying the question whether “randomness” is indeed necessary, one of the important questions of today’s complexity theory. It is intuitively clear that a good prg is sufficient for “derandomizing” all randomized algorithms efficiently. It is, however, quite difficult to find out precisely the type of pseudo randomness that is sufficient for derandomization and that can be derived from a reasonable complexity assumption. Here we review some of the recent important results.

Let us begin with a result based on one-way functions.

Theorem 16. (Håstad, Impagliazzo, Levin and Luby '99) If there exists a one-way function that cannot be invertible by any polynomial-size circuits, then we have

$$\text{BPP} \subseteq \text{SUBEXP} \stackrel{\text{def}}{=} \bigcap_{\delta > 0} \text{DTime}(2^{\ell^\delta}).$$

Remarks:

1. Here we need yet stronger noninvertibility, namely, *nonuniform* polynomial-time invertibility.
2. This consequence is important, because $\text{BPP} \subseteq \text{SUBEXP}$ implies $\text{BPP} \subsetneq \text{DEXP}$.

For the cryptographic use, the polynomial-time computability of prg is necessary, but it may not be so important for the derandomization. To see this point, let us clarify here what we want.

We would like to derandomize a given polynomial-time randomized program A , whose running time is bounded by some polynomial p . More specifically, we want to design some deterministic program simulating A , on a given input x , within time that is significantly smaller than $2^{p(|x|)}$. Consider any input x , and let ℓ be its length. Notice that A uses at most $p(\ell)$ random bits. Thus, for the simulation, a binary sequence of length ℓ that looks random to A is enough. Suppose that some prg g generates such a pseudo random

sequence from a seed of length $n = r(\ell) \ll \ell$ in $t(n)$ time. Then the standard simulation using this prg is to (1) generate pseudo random sequences by using g for all possible seeds $s \in \{0, 1\}^n$, and (2) execute A with these sequences and estimate the probability that A yields 1 on the input x . It is easy to see the total computation time is bounded as follows.

$$\text{time} \leq 2^n \cdot (t(n) + p(\ell)) = 2^{r(\ell)} \cdot (t(r(\ell)) + p(\ell)).$$

Notice that we do not have to keep t within polynomial; what is important, instead, is to choose r and t appropriately to minimize $2^{r(\ell)} \cdot t(r(\ell))$.

Recall our construction of prg. The key property for proving Theorem ?? (in particular, for the pseudo randomness) is the hardness of a hard-core predicate H , which is guaranteed by the hardness of h^{-1} (or f^{-1}). A similar construction is possible based on any hard function. In fact, by using the technique developed by Nisan and Wigderson '94, we can construct some prg based on the hardness assumption of DEXP; this prg is not polynomial-time computable, but sufficient for yielding the same simulation result.

Theorem 17. (Babai etal '93) If $\text{DEXP} \not\subseteq \text{P/poly}$, then $\text{BPP} \subseteq_{\text{i.o.}} \text{SUBEXP}$.

Remarks:

By $\text{BPP} \subseteq_{\text{i.o.}} \text{SUBEXP}$, we mean that every problem in BPP is solved by a deterministic program whose running time is $O(2^{\ell^\delta})$ for any δ for infinitely many input length ℓ . Certainly, this inclusion relation is much weaker than the ordinary relation $\text{BPP} \subseteq \text{SUBEXP}$. But on the other hand, if we indeed have $\text{BPP} \not\subseteq \text{SUBEXP}$, then we may expect that some BPP problem is hard for almost all input length, implying that even $\text{BPP} \subseteq_{\text{i.o.}} \text{SUBEXP}$ does not hold.

Impagliazzo and Wigderson '02 pushed this argument further to get the following, which is (almost) the best known result of this type.

Theorem 18. If $\text{DEXP} \not\subseteq \text{BPP}$, then $\text{BPP} \subseteq_{\text{i.o.}} \text{HeurSUBEXP}$.

Remarks:

1. The class HeurSUBEXP is the class of problems L for which there is a program A running SUBEXP time with the following property: (Intuitively) it is hard to find instances on which A errors. In other words, A gives a correct answer for “almost all” instances.
2. From the theorem, we now know either $\text{BPP} = \text{DEXP}$, or BPP is much smaller than DEXP , and it cannot be in between.

On the other hand, the following is best known assumption for deriving $\text{BPP} = \text{P}$.

Theorem 19. (Impagliazzo and Wigderson '97) If $\text{E} \not\subseteq_{\text{i.o.}} \text{SIZE}(2^{o(n)})$ (intuitively, E is hard enough even in the nonuniform computation model), then $\text{BPP} = \text{P}$.

3. Optimization Problem, Approximation, and PCP

Here again we use *our* definition of the class NP. Let L be any NP problem. Then we have some polynomial q_L and polynomial-time computable predicate R_L satisfying the following for all inputs x .

$$x \in L \iff \exists w \in \{0, 1\}^{q_L(|x|)} [R_L(x, w)].$$

We sometimes consider the quality of each witness. Note that, for a positive instance x , there may be more than one witnesses; time to time, we need the one with the best (or nearly best) quality. This is an (NP-type) optimization problem.

An optimization problem is formally defined in terms of q_L , R_L , and c_L , where c_L is called a *cost function*, specifying a cost of each witness. That is, for each (x, w) such that $R_L(x, w)$ holds, $c_L(x, w)$ gives the cost of w for x . We assume that c_L is also polynomial-time computable, and $c_L(x, w)$ is a nonnegative integer no more than $2^{e_L(|x|)}$ for some polynomial e_L . Notice that $c_L(x, w)$ is undefined if $R_L(x, w)$ does not hold. Define opt_L to be a function that gives the optimum cost for $x \in L$. That is,

$$\text{opt}_L(x) = \max\{ c_L(x, w) : R_L(x, w) \}.$$

Note that this is for maximization problems. For minimization problems, opt_L is defined similarly by using the min operator instead of max.

Definition 7. For any NP problem L , an *optimization problem* based on L (w.r.t. R_L (or L) and c_L) is to compute, for a given input x , one of the witnesses w achieving $c_L(x, w) = \text{opt}_L(x)$.

Remarks:

1. MAX-X or MIN-X or OPT-X is usually used to denote optimization problems, where X refers to the original NP problem. We also use OPT-NP to denote the class of these optimization problems.
2. When considering optimization problems, we often relax the witness condition (i.e., weaken a predicate R_L) so that all inputs become positive.

Example 3. MAX- k SAT is the following problem.

- Input:** A Boolean formula of the form $F = C_1 \wedge C_2 \wedge \dots \wedge C_m$,
where each C_i is a disjunction of k literals over X_1, \dots, X_n .
- Witness:** Any Boolean assignments $\vec{a} = (a_1, \dots, a_n)$ to X_1, \dots, X_n .
- Cost:** $c_{\text{SAT}}(F, \vec{a}) =$ the number of satisfied clauses of F .

Theorem 20. Every optimization problem is polynomial-time solvable by using some oracle in NP. That is, $\text{OPT-NP} \subseteq \Delta_2^{\text{P}} (= \text{P}^{\text{NP}})$.

Notations:

Recall $\Delta_2^{\text{P}} (= \text{P}^{\text{NP}})$ is a class of decision problems. So, the above usage is not correct, precisely speaking ;-)

Proof Idea. Consider any pair (R_L, c_L) defining some maximization problem MAX- L . First we claim that the opt_L is polynomial-time computable by using some NP oracle. In fact, the standard binary search algorithm computes opt_L by using the following NP problem.

$$L_{\text{cost}} = \{ (x, k) : x \in L \text{ and } k \leq \text{opt}_L(x) \}.$$

Once $\text{opt}_L(x)$ is obtained, the search for w satisfying $c_L(x, w) = \text{opt}_L(x)$ can be done, again, in polynomial-time by using the following oracle.

$$L_{\text{search}} = \{ (x, k, u) : \exists w \succ u [R_L(x, w) \wedge c_L(x, w) = k] \}.$$

• Approximability

It is easy to see that some optimization problems in OPT-NP is NP-hard and they are not exactly solvable in polynomial-time unless $\text{P} = \text{NP}$. But, on the other hand, there are many cases that approximate solutions are good enough, and in fact, some problems have reasonably good approximation algorithms. Thus the approximability of optimization problems is also an important subject in computational complexity theory.

For discussing the usefulness of various approximation algorithms, the following approximation measure is commonly used.

Definition 8. Consider any optimization problem OPT- L w.r.t. L and c_L and any algorithm A for OPT- L . For any $\epsilon > 0$, we say that A is an ϵ -approximation algorithm if for every x , A produces an ϵ -approximate solution w , a solution satisfying the following condition.

$$\frac{|c_L(x, w) - \text{opt}_L(x)|}{\max\{\text{opt}_L(x), c_L(x, w)\}} \leq \epsilon.$$

Remarks:

1. We may assume that $\epsilon \in (0, 1)$. Smaller ϵ is better.
2. This definition works for both maximization and minimization problems. Intuitively, the following equivalent statements may be easier to understand.

$$\begin{aligned} (\text{max.}) \quad c_L(x, w) &\geq (1 - \epsilon) \cdot \text{opt}_L(x). \\ (\text{min.}) \quad c_L(x, w) &\leq \frac{1}{1 - \epsilon} \cdot \text{opt}_L(x). \end{aligned}$$

3. For (mainly) discussing minimization problems, the quality of a solution w is estimated by using the following alternative ratio. Note that, for minimization problems, $1 \leq r(x, w) \leq r$ is equivalent to $c_L(x, w) \leq r \cdot \text{opt}_L(x)$.

$$r(x, w) = \max \left\{ \frac{c_L(x, w)}{\text{opt}_L(x)}, \frac{\text{opt}_L(x)}{c_L(x, w)} \right\}.$$

Definition 9. (Degree of Approximability)

- (1) The *approximation threshold* of OPT-X is the greatest lower bound of all $\epsilon > 0$ such that there is a polynomial-time ϵ -approximation algorithm for OPT-X.
- (2) We say that OPT-X has a *polynomial-time approximation scheme* (PTAS) if it has a polynomial-time ϵ -approximation algorithm for any $\epsilon > 0$.
- (3) We say that OPT-X has a *fully polynomial-time approximation scheme* (FPTAS) if it has a polynomial-time algorithm A satisfying the following for any given x and $\epsilon > 0$,
 - (i) $A(x, \epsilon)$ halts $p(|x| + 1/\epsilon)$ time, where p is some polynomial, and
 - (ii) $A(x, \epsilon)$ returns a ϵ -approximate solution for x .

Remarks:

1. The approximation threshold is *undefined* if OPT-X has PTAS.
2. Though the definition of PTAS allows us to define an algorithm for each ϵ , we would usually design one algorithm that works for all ϵ . The difference from FPTAS is that its running time cannot be bounded by polynomial in $1/\epsilon$.

Definition 10. (L-Reducibility: A tool for comparing approximability)

Consider any two maximization problems MAX-A and MAX-B that are specified by (R_A, c_A) and (R_B, c_B) respectively. We say that A is *L-reducible* to B if there are polynomial-time computable functions f and g and constants α and β satisfying the following.

- (a) For any input x of MAX-A, $y = f(x)$ is an instance of MAX-B that satisfies

$$\text{opt}_B(y) \leq \alpha \cdot \text{opt}_A(x).$$

- (b) For any $y = f(x)$ and any solution u of y (w.r.t. MAX-B), $v = g(u)$ is a solution of x that satisfies

$$\text{opt}_A(x) - c_A(x, v) \leq \beta \cdot (\text{opt}_B(y) - c_B(y, u)).$$

Proposition 21. Let MAX-A and MAX-B be any maximization problems such that MAX-A is L-reducible to MAX-B with reductions (f, g) and constants α, β . If MAX-B has a polynomial-time ϵ -approximation algorithm, then MAX-A has a polynomial-time $\frac{\alpha\beta\epsilon}{1-\epsilon}$ -approximation algorithm.

• **PCP and in-approximability**

Here we study PCP for our final complexity class, and study a recent beautiful result that provides us a surprising PCP characterization of the class NP. We will see that this result in fact yields the in-approximability of some optimization problems.

We start by introducing the notion of “probably checkable proof”. A *probably checkable proof* (in short, PCP) is a way to give a *proof* (or witness) that can be checked by a polynomial-time randomized *verifier* V . We assume that a proof Π is given in some special register and that a verifier V has a “random access” to every bit of Π ; that is, V can get the i th bit of Π in one step by specifying i . We say that V *accepts* Π if it halts with output 1, and *rejects* Π otherwise. Intuitively, V *accepts* Π if it is convinced by Π .

More formally, each PCP system is specified by a verifier V . For any problem L , we say that L *has PCP checkable by* V if the following holds. (Here ϵ can be any constant smaller than 1.)

$$\forall x \in \{0, 1\}^* \left[\begin{array}{l} x \in L \implies \exists \Pi \ [\Pr_V \{ V_G \text{ accepts } \Pi \} = 1] \\ x \notin L \implies \forall \Pi \ [\Pr_V \{ V_G \text{ accepts } \Pi \} \leq \epsilon] \end{array} \right]$$

Definition 11. For any functions r and q , the class $\text{PCP}(r, q)$ consists of all problems L that has PCP checked by some polynomial-time randomized verifier which, on any input of length ℓ , uses $O(r(\ell))$ random bits and examines $O(q(\ell))$ bits of a given proof Π .

Remarks:

Though not specified, we assume that the size of each proof Π is determined from input length ℓ . For some PCP, this length could become exponential in ℓ . even if its verifier is polynomial-time. But in the case that $r(\ell) = O(\log \ell)$, we may assume that $|\Pi|$ is polynomially bounded w.r.t. ℓ .

It is easy to see that the idea of PCP is a generalization of our definition of NP. In fact, the following relation is easy from the definition.

Proposition 22. $\text{NP} \subseteq \cup_{p:\text{poly}} \text{PCP}(0, p)$.

What is surprising is the following characterization, which is due to Arora, Lund, Motwani, Sudan, and Szegedy '92.

Theorem 23. $\text{NP} \subseteq \text{PCP}(\log \ell, 1)$.

Now we explain how this characterization is used for some inapproximability analysis.

To be concrete, let us suppose that some NP-complete problem, say, 3COL (3-Coloring problem), has a PCP verifier V that can check a given proof Π for an input of length ℓ using $r(\ell)$ random bits and looking at exactly q bits of Π with error probability bounded by ϵ . Then we can prove the following inapproximability.

Theorem 24. The above assumption implies that the approximation threshold of MAX- q SAT is at least $(1 - \epsilon)2^{-q}$ unless $P = NP$.

Proof Idea. Let G be any input graph of size ℓ for the 3-Coloring problem, and let Π be any PCP proof for G . The proof may not be correct, and even G may not be three colorable.

Consider the execution of V_G on Π . From our assumption, V uses $r(\ell)$ random bits and examine q bits of Π . Furthermore, we may assume that these q bit positions that V_G checks are completely determined by $r(\ell)$ random bits. (That is, V_G does not change the bit positions depending on Π .)

Let ρ be a sequence of length $r(\ell)$ denoting the random bits used in the execution of V_G , and let $v(\rho, 1), \dots, v(\rho, q)$ be the bit positions examined by V_G . Then the decision of V_G is determined by ρ and the bits $\Pi_{v(\rho,1)}, \dots, \Pi_{v(\rho,q)}$, where Π_k denotes the k th bit of Π . For example,

$$\begin{aligned} V_G \text{ rejects } \Pi \text{ on } \rho &\iff [(\Pi_{v(\rho,1)} = 0) \wedge (\Pi_{v(\rho,2)} = 0) \wedge \dots \wedge (\Pi_{v(\rho,q)} = 0)] \\ &\vee [(\Pi_{v(\rho,1)} = 0) \wedge (\Pi_{v(\rho,2)} = 1) \wedge \dots \wedge (\Pi_{v(\rho,q)} = 1)] \\ &\vee [(\Pi_{v(\rho,1)} = 1) \wedge (\Pi_{v(\rho,2)} = 0) \wedge \dots \wedge (\Pi_{v(\rho,q)} = 0)]. \end{aligned}$$

Then

$$\begin{aligned} V_G \text{ accepts } \Pi \text{ on } r &\iff [(\Pi_{v(\rho,1)} = 1) \vee (\Pi_{v(\rho,2)} = 1) \vee \dots \vee (\Pi_{v(\rho,q)} = 1)] \\ &\wedge [(\Pi_{v(\rho,1)} = 1) \vee (\Pi_{v(\rho,2)} = 0) \vee \dots \vee (\Pi_{v(\rho,q)} = 0)] \\ &\wedge [(\Pi_{v(\rho,1)} = 0) \vee (\Pi_{v(\rho,2)} = 1) \vee \dots \vee (\Pi_{v(\rho,q)} = 1)]. \end{aligned}$$

Now we regard each Π_k as a Boolean variable; then the righthand side of the above is the conjunction of three clauses each containing q literals. In this way, we can define a CNF formula F_ρ characterizing whether V_G yields 1 on ρ , for each binary sequence ρ . Let F_G be a conjunction of all F_ρ 's. Note that the number of clauses in F_G , which we denote by m , is at most $2^q \cdot 2^{r(\ell)}$.

It is not so hard to see that $G \in 3\text{COR}$ implies that there must be some assignment to all Π_k 's that satisfy all clauses; namely, the PCP proof for G . On the other hand, if $G \notin 3\text{COR}$, then we can show that there are at least $(1 - \epsilon)2^{r(\ell)}$ unsatisfied clauses for any assignment, since V_G rejects every Π with probability at least $1 - \epsilon$. That is, we have

$$\begin{aligned} G \in 3\text{COR} &\implies \text{opt}_{\text{SAT}}(F_G) = m, \\ G \notin 3\text{COR} &\implies \text{opt}_{\text{SAT}}(F_G) = m - (1 - \epsilon)2^{r(\ell)}. \end{aligned}$$

Now suppose that, for some $\gamma < (1 - \epsilon)2^{-q}$, we have some polynomial-time γ -algorithm for MAX- q SAT. Then for any $G \in 3\text{COR}$, the algorithm yields some solution Π that satisfies at least $m_1 = (1 - \gamma)m = m - \gamma \cdot m$ clauses. On the other hand, for any $G \notin$

3COR, any solution satisfies at most $m_2 = m - (1 - \epsilon)2^{r(\ell)}$ clauses. But from our choice of γ , we have

$$(1 - \epsilon)2^{r(\ell)} < \gamma \cdot 2^q \cdot 2^{r(\ell)} = \gamma \cdot m.$$

That is, $m_1 > m_2$. Therefore, we can decide whether $G \in 3COR$, by running the algorithm on F_G and checking whether the solution satisfies more than m_2 clauses. That is, 3COR is polynomial-time solvable!