Computational Complexity Theory

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

Part I Basics

0. Introduction — What Shall We Study?

In this course, I am planning to cover the following topics.

Part I Basics

1. Problem and Algorithm
2. Computation Model and Complexity Measure
3. Hierarchy Theorem
4. Some Standard Complexity Classes
5. Reducibility and Completeness
6. Some Other Complexity Classes

Part II Some Advanced Topics

7. Polynomial-Time Hierarchy and Interactive Proof
8. One-Way Function and Pseudo Random Sequence Generator
9. Optimization Problem, Approximation, and PCP

References

[1] M. Sipser, Introduction to the Theory of Computation, PWS Pub., 1997.
[2] J. Van Leeuwen, ed., Handbook of Theoretical Computer Science, Vol.A: Algorithms and Complexity, Elsevier, 1990.
[3] R. Motwani and P. Raghavan, Randomized Algorithms, Cambridge Univ. Press, 1995.
[4] G. Ausiello, etal, Complexity and Approximation, Springer, 1999.

1. Problem and Algorithm

Before starting technical explanation, let us first agree on a framework that is assumed common to computational complexity theory.

• What is our problem?

In computational complexity theory, we consider only problems that are completely and formally specified. But even among mathematically specified problems, a problem like "Solve Riemann's conjecture" is not of our interest. Instead, we consider a problem asking for an answer to a given *instance* of the problem, i.e.,a problem of computing some given function on a given input. For example, a problem is specified in the following way:

> Max. Clique Problem (abbrev. CLIQUE)
> **Instance**: An undirected graph G.
> **Question**: What is the size of the largest clique (i.e., complete subgraph) of G?

Remarks:

1. Input instances and outputs are encoded appropriately as binary strings, i.e., finite strings over the alphabet $\{0, 1\}$.
2. In our *general* discussion, the *size* of an input is simply its length. Throughout this course, we use ℓ to denote the input size.

• Decision problem

Among computational problems like CLIQUE, we consider only "decision problems", where a *decision problem* is to decide whether a given instance is 'yes' (or simply 1) or 'no' (simply 0). For example, instead of the CLIQUE problem mentioned above we can consider the following decision version.

> Max. Clique Problem (CLIQUE)
> **Instance**: An undirected graph G and an integer k.
> **Question**: Does G have a clique of size $\geq k$?

Remarks:

1. These two versions of CLIQUE have almost the same complexity. By using the standard binary search strategy, we can easily solve the previous CLIQUE by using any algorithm for its decision version. This type of relationship always holds in general.
2. Notice, however, there is some subtle difference, which becomes essential if we want to discuss much finer complexity issue. We will see such an example later.

3. Any decision problem is completely specified by the set of 'yes' instances of the problem, or more precisely, the set of binary strings encoding those 'yes' instances. Thus, we usually identify a problem with its 'yes' instances. For example,

$$\text{CLIQUE} \;=\; \{\ (G,k)\ :\ G \text{ has a clique of size} \geq k\ \}.$$

• Function and algorithm

A problem can be regarded as a function mapping $\{0,1\}^*$ to $\{0,1\}^*$, and an *algorithm* (for the problem) is a method for computing this function. We assume that each algorithm is implemented by a *program* for some machinery, and we will identify them throughout this course.

Notations:

function:	f, g_{clique}, ...	usage:	$f(x) \Leftarrow$ the value of f on x.
program:	`A`, `clique`, ...	usage:	$\texttt{A}(x) \Leftarrow$ the output of `A` on x *or* the execution of `A` on x.

We say that an algorithm `A` (or, a program `A`) *solves* a decision problem L if the following holds.

$$\forall x \in \{0,1\}^* \quad \left[\begin{array}{rcl} x \in L & \Longrightarrow & \texttt{A}(x) = 1 \\ x \notin L & \Longrightarrow & \texttt{A}(x) = 0 \end{array}\right]$$

2. Computation Model and Complexity Measure

For our basic computation model, we consider some standard computation device and assume some simple programming language controlling the device. For example, we consider the following program.

```
program A
input register:  $1
output register: $1
work registers:  $2, $3, $4
1: $2 ← ···     ⇐ Only very primitive instructions are allowed.
2: if $2 ≠ 0 then goto 6
3:    ⋮
4:    ⋮
5: goto 2
6: halt
```

Remarks:

1. Though we omit details, there are two choices for our model: a register contains *either* a binary string, *or* an integer (in binary). In each choice, we assume that each register can keep a string or an integer of any length.
2. We assume that instructions are simple but strong enough to implement any algorithm. For example, there is no single instruction for the comparison of two registers; such comparison should be implemented by a sequence of instructions.
3. We assume that each program is encoded by a binary string. For example, `A` is encoded as

$$\langle \$1, \$1, \langle \$2, \$3, \$4 \rangle, s_1, s_2, ..., s_6 \rangle, \quad (\Leftarrow \text{this can be encoded in } \{0,1\}^*)$$

   where s_1, ..., s_6 are codes for the statements.

• **Time and space complexity measures**

Consider any program $\mathtt{A}$. For a given input x, we estimate its *computation time* and *consumed space* as follows.

$$\begin{aligned} \mathrm{time}_{\mathtt{A}}(x) &= \text{the number of executed instructions until } \mathtt{A}(x) \text{ halts.} \\ \mathrm{space}_{\mathtt{A}}(x) &= \text{the total number of bits in all registers.} \end{aligned}$$

Since we are interested in how the complexity grows when input size gets increased, our complexity analysis is based on the following functions, which are respectively called a *time complexity* and a *space complexity*.

$$\begin{aligned} \mathrm{time}_{\mathtt{A}}(\ell) &= \max\{\ \mathrm{time}_{\mathtt{A}}(x) \,|\, x \in \{0,1\}^{\ell}\ \}. \\ \mathrm{space}_{\mathtt{A}}(\ell) &= \max\{\ \mathrm{space}_{\mathtt{A}}(x) \,|\, x \in \{0,1\}^{\ell}\ \}. \end{aligned}$$

Remarks:

1. These complexity measures are called *worst-case complexity measures*. Clearly, the worst-case criteria is not our only choice, and it is not always the best choice either. For example, there are cases that the average-case analysis is more appropriate. But the average-case analysis is usually complex, and it is even more difficult to provide a simple and appropriate framework for studying the average-case complexity; so, average-case complexity has not been studied until recently.
2. When we discuss space complexity smaller than ℓ, the space used for input and output registers is not counted. But in this case, input and output registers should be used as read-only and write-only registers.

• **Big-O notation**

Definition 1. For any functions f and g from $\mathbf{Z}^+$ to $\mathbf{Z}^+$, we write $f = O(g)$ if the following holds for some constants c_0 and n_0.

$$\forall n \geq n_0 \; [\; f(n) \;\leq\; c_0 \cdot g(n) \;].$$

Remarks:

1. The big-O notation does not imply the asymptotic equivalence between two functions; it just means that one function is asymptotically bounded *from above* by another function. For example, we can write $f = O(g)$ for $f(n) = n$ and $g(n) = 2^n$.
2. Important examples: (below p denotes any polynomial)

$$\begin{array}{l} p(\log n) \;=\; O((\log n)^k) \;=\; O(n) \quad (\text{for some } k > 0). \\ p(n) \;=\; O(n^k) \;=\; O(2^n) \quad (\text{for some } k > 0). \\ f(n) = O(g(n)) \;\&\; g(n) = O(h(n)) \implies f(n) = O(h(n)). \end{array}$$

• **Basic complexity classes**

Let t and s be any functions on natural numbers.

$$\begin{array}{rcl} \mathrm{DTime}(t) & = & \{\; L \;:\; L \text{ is solvable by some } \mathtt{A} \text{ whose time complexity is } O(t) \;\}. \\ \mathrm{DSpace}(s) & = & \{\; L \;:\; L \text{ is solvable by some } \mathtt{A} \text{ whose space complexity is } O(s) \;\}. \end{array}$$

These are basic complexity classes. The symbol "D" denotes that they are ordinary deterministic complexity measures.

Functions used for t (resp., s) is called a *time bound* (resp., *space bound*).

Proposition 1. For any t_1 and t_2, and s_1 and s_2, we have

$$\begin{array}{rcl} t_1 = O(t_2) & \implies & \mathrm{DTime}(t_1) \;\subseteq\; \mathrm{DTime}(t_2), \\ s_1 = O(s_2) & \implies & \mathrm{DSpace}(s_1) \;\subseteq\; \mathrm{DTime}(s_2). \end{array}$$

Proof. For any problem L in $\mathrm{DTime}(t_1)$, we can show that it is indeed in $\mathrm{DTime}(t_2)$ as follows. Thus, $\mathrm{DTime}(t_1) \subseteq \mathrm{DTime}(t_2)$.

$$\begin{array}{rcl} L \in \mathrm{DTime}(t_1) & \implies & \text{some } \mathtt{A} \text{ solves } L \text{ and } \mathrm{time}_{\mathtt{A}} = O(t_1) \\ & \implies & \mathrm{time}_{\mathtt{A}} = O(t_2) \implies L \in \mathrm{DTime}(t_2). \end{array}$$

3. Hierarchy Theorem

Consider any time bounds t_1 and t_2. If $t_1 << t_2$, we may naturally expect $\mathrm{DTime}(t_1) \subsetneq \mathrm{DTime}(t_2)$. This intuition is provable — *Hierarchy Theorem*.

• Interpreter and its efficiency

Consider the following functions.

$$\begin{aligned}
\mathit{IsProgram}(a) &= [\,a \text{ is a correct code of some program with one input.}\,] \\
\mathit{eval}(a,x) &= \begin{cases} a(x), & \text{if } \mathit{IsProgram}(a), \\ 0, & \text{otherwise.} \end{cases} \\
\mathit{eval\text{-}in\text{-}time}(a,x,\bar{t}) &= \begin{cases} a(x), & \text{if } \mathit{IsProgram}(a) \text{ and } a(x) \text{ halts in } t \text{ steps}, \\ 0, & \text{otherwise.} \end{cases}
\end{aligned}$$

Notations:
Recall that a program is encoded by a binary string a. By $a(x)$, we mean the output (or, in more general, the execution) of the program coded by a on x. For any integer t, we use $\bar{t}$ to denote its unary notation, that is, $\bar{t} = 0^t = \underbrace{00\cdots0}_{t}$.

Remarks:

1. We may assume that there are programs computing these functions. Let us denote them `IsProg`, `eval`, and `evt`. `IsProg` is a *syntax checker*, and `eval` is an *interpreter*.
2. We may assume that these programs are reasonably efficient. In particular, with some constants $c_{\mathtt{evt}}$ and $d_{\mathtt{evt}}$, we assume that the following bound holds for any a, x, and $t \geq |x|$.

$$\begin{aligned}
\mathrm{time}_{\mathtt{evt}}(a,x,\bar{t}) &\leq c_{\mathtt{evt}}|a|t^2 + d_{\mathtt{evt}}. \\
\mathrm{space}_{\mathtt{evt}}(a,x,\bar{t}) &\leq c_{\mathtt{evt}}|a|t + d_{\mathtt{evt}}.
\end{aligned}$$

Question?!

Q1. Why does `evt` need such running time? Why not $O(|a|t)$ or even $O(|a|+t)$?

• **In-computability**

Consider the problem of computing the following function, which we will refer as DIAG. (Note that (a, x) is again encoded in $\{0,1\}^*$. Thus, DIAG is an ordinary decision problem.)

$$\mathit{diag}((a,x)) \;=\; [\,\mathit{eval}(a,(a,x)) \neq 1\,].$$

Theorem 2. The problem DIAG is not solvable.

Proof. Enumerate of all program codes as a_1, a_2, ... Let α_i be the output of a_i on input $(a_i, 0)$. That is,

$$\alpha_i \;=\; a_i((a_i,0)) \;=\; \mathit{eval}(a_i,(a_i,0)).$$

Hence, by *definition*, we have $\mathit{diag}((a_i,0)) \neq \alpha_i$; that is, each a_i disagrees with *diag* on $(a_i, 0)$. Therefore, no program a_k computes *diag*.

• **Time and space hierarchy theorems**

Theorem 3. (Given first by Hartmanis and Stearns '65)
Let t_1 and t_2 be any "reasonable" bounds such that $t_1 = O(t_2)$.
(1) $t_2 \neq O(t_1^2) \Longrightarrow \mathrm{DTime}(t_1) \subsetneq \mathrm{DTime}(t_2)$.
(2) $s_2 \neq O(s_1) \Longrightarrow \mathrm{DTime}(s_1) \subsetneq \mathrm{DTime}(s_2)$.

Remarks:

1. Hartmanis and Stears proved a slightly different version of time hierarchy theorem. The space hierarchy theorem was proved first by Hartmanis, Lewis, and Stearns.
2. The quality of the theorem depends on the efficiency of an interpreter. For example, a linear time interpreter would yield a separation like the space hierarchy theorem.
3. Similar separation results hold for all reasonable complexity classes, in particular, those we will define later.

• **An example proof**

To see how the theorem is proved, let us prove $\mathrm{DTime}(\ell) \subsetneq \mathrm{DTime}(\ell^4)$.

For this, consider the following *new version* of *diag*. (Again we use DIAG for the problem computing *diag*.)

$$\begin{array}{l} \mathit{diag}((a,x)) \;=\; [\,\mathit{eval\text{-}in\text{-}time}(a,(a,x),\overline{t}) \neq 1\,], \\ \text{where } \ell = |(a,x)| \text{ and } t = \ell^2/|a|. \end{array}$$

Then we can show that DIAG is (i) solvable in $O(\ell^4)$ time, but (ii) not solvable in $O(\ell)$. That is, it witnesses $\mathrm{DTime}(\ell^4) - \mathrm{DTime}(\ell) \neq \emptyset$.

(i) $\mathrm{DIAG} \in \mathrm{DTime}(\ell^4)$.
We can simply use our interpreter `evt` to compute *diag*. Its computation time is bounded as follows. (Let $\ell = |(a,x)|$, the input length.)

$$\begin{array}{rcl} \text{time} & \leq & \text{time}_{\mathtt{evt}}(a,x,\overline{t}) + O(\ell^2) \;\leq\; c_{\mathtt{evt}}|a|t^2 + d_{\mathtt{evt}} + O(\ell^2) \\ & = & c_{\mathtt{evt}}|a|\left(\dfrac{\ell^2}{|a|}\right)^2 + d_{\mathtt{evt}} + O(\ell^2) \;\leq\; c\ell^4 + d \;=\; O(\ell^4). \end{array}$$

(ii) $\mathrm{DIAG} \notin \mathrm{DTime}(\ell)$.
Consider now the enumeration a_1, a_2, ... of all programs whose running time is $O(\ell)$. More precisely, we assume that the time complexity of each a_i is bounded by $c_i\ell + d_i$ for some c_i and d_i.

For each a_i (i.e., each c_i, d_i, and $|a_i|$), choose n_i satisfying the following, where $\ell_i = |(a, 0^{n_i})|$.

$$c_i\ell_i + d_i \;\leq\; \frac{\ell_i^2}{|a_i|}.$$

Such n_i certainly exists, because $\ell^2 \neq O(\ell)$.

Now define $\alpha_i = a_i((a_i, 0^{n_i}))$. Then we have (letting $\ell_i = |(a_i, 0^{n_i})$ and $t_i = \ell_i^2/|a_i|$)

$$\begin{array}{rcl} \alpha_i & = & a_i((a_i, 0^{n_i})) \\ & = & \mathit{eval}(a_i, (a_i, 0^{n_i})) \\ & = & \mathit{eval\text{-}in\text{-}time}(a_i, (a_i, 0^{n_i}), \overline{t_i}). \end{array}$$

Hence, a_i disagrees with *diag* on $(a_i, 0^{n_i})$, and therefore, no a_k computes *diag* as before.

The last equality in the above holds from the following reason: From our choice of n_i, t_i is larger than the time bound for $a_i(a_i, 0^{n_i})$. Hence having time bound t_i does not affect the output of the eval.

4. Some Standard Complexity Classes

What follows is a summary of well known complexity classes. Here we explain classes in the first two rows.

comp. model	time	space
deterministic	P, DEXP	DLOG, PSPACE
nondet.	NP, co-NP, NEXP	NLOG
randomized	ZPP, RP, co-RP, BPP, PP	RLOG
circuits	AC^0, NC, P/poly	

Table 1: Popular complexity classes

• Deterministic complexity classes

Time classes:

$$\mathrm{P} = \bigcup_{p:\mathrm{poly}} \mathrm{DTime}(p), \quad \mathrm{E} = \bigcup_{c>0} \mathrm{DTime}(2^{c\ell}), \quad \mathrm{DEXP} = \bigcup_{p:\mathrm{poly}} \mathrm{DTime}(2^{p(\ell)}).$$

Space classes:

$$\mathrm{DLOG} = \mathrm{DSpace}(\log \ell), \quad \mathrm{PSPACE} = \bigcup_{p:\mathrm{poly}} \mathrm{DSpace}(p).$$

Proposition 4.
(1) $\mathrm{P} \subseteq \mathrm{E} \subseteq \mathrm{DEXP}$.
(2) $\mathrm{DLOG} \subseteq \mathrm{PSPACE}$.

Proposition 5. $\mathrm{DLOG} \subseteq \mathrm{P} \subseteq \mathrm{PSPACE} \subseteq \mathrm{DEXP}$.

Theorem 6.
(1) $\mathrm{P} \subsetneq \mathrm{E} \subsetneq \mathrm{DEXP}$.
(2) $\mathrm{DLOG} \subsetneq \mathrm{PSPACE}$.

Remarks:

1. DLOG (similarly, NLOG and RLOG) are often abbreviated as DL.
2. Relations in Proposition **??** are immediate from the definition. Proposition **??** is also easy, considering the relation between time and space bounds. On the other hand, the separations in Theorem **??** are from the hierarchy theorems.
3. We cannot prove, e.g., $\mathrm{P} \subsetneq \mathrm{E}$, by simply showing $\mathrm{DTime}(\ell^k) \subsetneq \mathrm{DTime}(2^\ell)$. (For example, $[0, 1-(1/k)] \subsetneq [0,1)$ for all $k \geq 1$. But we have $\cup_{k\geq 1}[0, 1-(1/k)] = [0,1)$.)

Question?!

Q2. Then how to prove $\mathrm{P} \subsetneq \mathrm{E}$?

• **The class NP**

We define the class NP in the following way (which is my favorite definition ;−).

Definition 2. NP is the class of problems L for which we can define some polynomial q_L and some polynomial-time computable predicate R_L with the following condition, which we will refer as the *NP-condition*.

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

Sets with this property are called *NP sets*. A string w like above is called a *witness* (of x being in L), and R_L is called a *witness checking predicate*.

Definition 3. Class co-NP is the class of problems whose compliment belong to NP.

Remarks:

1. Recall that we identify a decision problem L with a set of 'yes' instances. The compliment of a problem L is the problem specified by the set compliment of L. That is, the compliment of a problem L is the problem obtained by exchanging all 'yes' and 'no' answers of L.
2. The NP-condition is equivalent to the following one.

$$\forall x \in \{0,1\}^* \quad \left[\begin{array}{l} x \in L \implies \exists w : |w| \leq q_L(|x|) \; [\; R_L(x,w) = 1 \;] \\ x \notin L \implies \forall w : |w| \leq q_L(|x|) \; [\; R_L(x,w) = 0 \;] \end{array} \right]$$

3. On the other hand, any co-NP problem L has q_L and R_L satisfying the following.

$$\forall x \in \{0,1\}^* \quad \left[\begin{array}{l} x \in L \implies \forall w : |w| \leq q_L(|x|) \; [\; R_L(x,w) = 1 \;] \\ x \notin L \implies \exists w : |w| \leq q_L(|x|) \; [\; R_L(x,w) = 0 \;] \end{array} \right]$$

Example 1.

$$\begin{array}{lcl} \text{HAM} & = & \{\; G \;:\; G \text{ is a Hamiltonian graph } \}. \\ \text{COR} & = & \{\; (G,k) \;:\; G \text{ is } k\text{-colorable } \}. \\ \text{VC} & = & \{\; (G,k) \;:\; G \text{ has a vertex cover of size } k \;\}. \\ \text{SAT} & = & \{\; F \;:\; F \text{ is a satisfiable (extended) Boolean formula } \}. \\ \text{3SAT} & = & \{\; F \;:\; F \text{ is a satisfiable 3CNF formula } \}. \\ \text{KNAP} & = & \{\; (\,(s_1,...,s_n),(c_1,...,c_n),B,K\,) \;: \\ & & \quad \exists U \subseteq \{1,...,n\} \; [\; \sum_{u \in U} s_u \leq B \;\wedge\; \sum_{u \in U} c_u \geq K \;] \;\}. \\ \text{SSUM} & = & \{\; (\,(a_1,...,a_n),B\,) \;:\; \exists U \subseteq \{1,...,n\} \; [\; \sum_{u \in U} a_u = B \;] \;\}. \end{array}$$

Example 2.

$$\begin{array}{lcl}
\text{EULER} & = & \{\ G\ :\ G \text{ is an Eulerian graph }\}. \\
\text{LP} & = & \{\ (A, \mathbf{b}, \mathbf{c}, z)\ :\ \exists \mathbf{x} \geq \mathbf{0}\ [\ A\mathbf{x} \leq \mathbf{b}\ \wedge\ \mathbf{cx} \geq z\ ]\ \}. \\
\text{PRIME} & = & \{\ n\ :\ n \text{ is a prime number }\}. \\
\text{COMPO} & = & \{\ n\ :\ n \text{ is a composite number }\}. \\
\text{GI} & = & \{\ \langle G_1, G_2 \rangle\ :\ G_1 \text{ is isomorphic to } G_2\ \}.
\end{array}$$

Remarks:

1. Examples in Example **??** are somewhat easier than those in Example **??**. In fact, EULER and LP are polynomial-time solvable. This does not contradicts the notion of NP. Indeed, as shown below NP contains the class P.
2. In the definition of PRIME and COMPO, we assume the binary encoding of numbers. PRIME and COMPO are also in co-NP. It is easy to see that PRIME is in co-NP and COMPO is in NP. On the other hand, PRIME $\in$ NP (similarly, COMPO $\in$ co-NP) is not so trivial.
3. Currently, we can locate GI in NP $\cap$ co-AM, where co-AM is a randomized version of co-NP.

We can prove the following relationships.

Theorem 7.
(1) P $\subseteq$ NP $\cap$ co-NP.
(2) NP $\cup$ co-NP $\subseteq$ PSPACE ($\subseteq$ DEXP from Proposition **??**).

Proof. The relation (1) is immediate from the definition, and (2) is proved by a standard simulation. Here we show that NP $\subseteq$ PSPACE.

Let L be any NP problem. Thus, we have some q_L and R_L that satisfy the NP-condition for L; moreover, for R_L, we may assume some program `IsR` computing R_L in polynomial time. Then we can design the following algorithm `IsL` for L, and it is easy to see that it uses space bounded by $O(s_{\texttt{IsR}}(\ell + q_L(\ell)))$, which is polynomial in ℓ, where $s_{\texttt{IsR}}$ is a polynomial space bound for `IsR`.

```
program IsL(input x);
   q ← q_L(|x|);
   for each w ∈ {0,1}^{≤q} do {
      if IsR(x, w) = 1 then halt(1);
   }
   halt(0);
end-program.
```

• A nondeterministic computation model

Although many researchers have been using our definition of NP these days, there is an alternative definition of NP, which is useful (even necessary) for defining the other related complexity classes. This definition of NP requires a new computation model for *nondeterministic computation*.

We can give a nondeterministic computation model by extending our computation device and its programming language. More specifically, we introduce if-statements of the following type.

$$\textbf{if guess} = \texttt{0} \quad \textbf{then goto } \mathit{label}_1 \textbf{ else goto } \mathit{label}_2$$

The value of *guess* is either `0` or `1`, and the following computation may differ depending on this value. When the execution reaches to this statement, the machine executes both possibilities *in parallel*.

A *nondeterministic program* is a program having this type of special if-statements. For defining nondeterministic computation, we need to define the meaning (i.e., the semantics) of such nondeterministic programs.

Let `A` be a nondeterministic program. For any input x to `A`, a *computation path of* `A` *on* x is a computation of `A` on x following one sequence of guesses. We regard that `A` outputs `1` on x (or more clearly, we say `A` *accepts* x) if `A` yields the output 1 on one of its computation paths. Otherwise (i.e., if `A` on x has no such computation path), we regard that `A` outputs 0 on x (or more clearly, `A` *rejects* x). We say that `A` *solves* a problem L if its accepts x if and only if $x \in L$.

The time and space complexity can be measured in the same way as before. We introduce the following basic complexity classes.

$$\begin{aligned} \mathrm{NTime}(t) &= \{\ L \ :\ L \text{ is solvable by some nondet. } \texttt{A} \text{ whose time complexity is } O(t)\ \}. \\ \mathrm{NSpace}(s) &= \{\ L \ :\ L \text{ is solvable by some nondet. } \texttt{A} \text{ whose space complexity is } O(s)\ \}. \end{aligned}$$

Then we can define the following standard nondeterministic complexity classes, including the class NP.

$$\begin{aligned} \mathrm{NP} &= \bigcup_{p:\mathrm{poly}} \mathrm{NTime}(p), & \mathrm{NEXP} &= \bigcup_{p:\mathrm{poly}} \mathrm{NTime}(2^{p(\ell)}). \\ \mathrm{NLOG} &= \mathrm{NSpace}(\log \ell), & \mathrm{NPSPACE} &= \bigcup_{p:\mathrm{poly}} \mathrm{NSpace}(p). \end{aligned}$$

Remarks:
Compliment classes such as co-NLOG and co-NEXP are defined as co-NP. For example, co-NLOG is the class of problems whose compliment is in NLOG.

Theorem 8. The above definition of NP is equivalent to ours.

• **Relation between det. and nondet. classes**

Proposition 9. DLOG $\subseteq$ NLOG, P $\subseteq$ NP, and DEXP $\subseteq$ NEXP.

Theorem 10. NLOG $\cup$ co-NLOG $\subseteq$ P, and NP $\cup$ co-NP $\subseteq$ PSPACE.

Theorem 11. (Savitch '70) PSPACE = NPSPACE

Theorem 12. (Immermann '88, Szelepcsényi '87) NLOG = co-NLOG.

Remarks:

1. Proposition **??** is immediate noticing that the nondet. ccomputation model is an extension of the deterministic one. (Thus, clearly, we have DTime(t) $\subseteq$ NTime(t) and DSpace(s) $\subseteq$ NSpace(s).)
2. Proposition **??** can be proved easily like the proof of Theorem **??**.
3. From Theorem **??**, we do not need the complexity class NPSPACE. In more general, it can be shown that NSpace(s) $\subseteq$ DSpace(s^2).
4. For deterministic classes such as P, we clearly have P = co-P; that is, deterministic complexity classes are closed under taking compliment. On the other hand, it is not clear at all whether nondeterministic classes also have this property. Theorem **??** states that NLOG do have this property. The theorem is proved by a quite interesting census argument.

5. Reducibility and Completeness

• Polynomial-time many-one reducibility

Definition 4. For any decision problems (or, sets of strings) A and B, we say that A is *many-one reducible* to B (abbrev. $A \leq_{\mathrm{m}} B$) if there exists a function h with the following properties. (h itself is called a *many-one reduction* (abbrev. $\leq_{\mathrm{m}}$-reduction).)
(a) h is a total function from $\{0,1\}^*$ to $\{0,1\}^*$,
(b) $\forall x \in \{0,1\}^*$ [$x \in A \iff h(x) \in B$], and
(c) h is computable.
If furthermore, h is polynomial-time computable, then we say that A is *polynomial-time many-one reducible* (abbrb. $A \leq_{\mathrm{m}}^{\mathrm{p}} B$). The *log-space many-one reducibility* (abbrb. $\leq_{\mathrm{m}}^{\log}$-reducibility) and any other resource-bounded reducibilities are defined similarly.

Example 3. The following set CLIQUE is $\leq_{\mathrm{m}}^{\mathrm{p}}$-reducible to IS. Conversely, IS is also $\leq_{\mathrm{m}}^{\mathrm{p}}$ to CLIQUE. (In fact, they are $\leq_{\mathrm{m}}^{\log}$-reducible each other.)

$$\begin{aligned} \text{CLIQUE} &= \{\ \langle G,k\rangle\ :\ G \text{ has a clique of size } k\ \}. \\ \text{IS} &= \{\ \langle G,k\rangle\ :\ G \text{ has an independent set of size } k\ \}. \end{aligned}$$

The following is the key property of $\leq_{\mathrm{m}}^{\mathrm{p}}$-reducibility.

Theorem 13.
(1) If $A \leq_{\mathrm{m}}^{\mathrm{p}} B$ and B is in P then A is also in P.
(2) The same relation holds by replacing P with its super-classes such as NP, co-NP, DEXP, ..., etc.

Proof. We prove (1). Suppose that $A \leq_{\mathrm{m}}^{\mathrm{p}} B$ by a reduction h, and B is solved by a program `IsB`. Then we can design polynomial-time program `IsA` solving A as follows.

```
program IsA(input x);
   y ← h(x);
   halt(IsB(y));
end-program.
```

Remarks:
There are various "polynomial-time" reducibility notions, with which we can prove the above theorem (or, more precisely, the statement (1) of the theorem). The many-one reducibility is in a sense the simplest. But we will mainly discuss with the many-one reducibility, and "many-one" is usually omitted in the following.

From this theorem, we can discuss the relative hardness of two problems by using $\leq^{\mathrm{p}}_{\mathrm{m}}$-reducibility. In fact, if $A \leq^{\mathrm{p}}_{\mathrm{m}} B$, then we have $B \in \mathrm{P} \Longrightarrow A \in \mathrm{P}$, $B \in \mathrm{DEXP} \Longrightarrow A \in \mathrm{DEXP}$, etc.; hence, we may consider that B's complexity bounds A's complexity, or A is no harder than B. That is,

$$A \leq^{\mathrm{p}}_{\mathrm{m}} B \overset{\text{intuitively}}{\Longrightarrow} \boxed{A\text{'s hardness} \leq B\text{'s hardness}}$$

Suppose that both $A \leq^{\mathrm{p}}_{\mathrm{m}} B$ and $B \leq^{\mathrm{p}}_{\mathrm{m}} A$ hold. Then we write $A \equiv^{\mathrm{p}}_{\mathrm{m}} B$, meaning that they are of the same hardness. Note also that $\leq^{\mathrm{p}}_{\mathrm{m}}$-reducibility is a pre-order.

Theorem 14. If $A \leq^{\mathrm{p}}_{\mathrm{m}} B$ and $B \leq^{\mathrm{p}}_{\mathrm{m}} C$, then $A \leq^{\mathrm{p}}_{\mathrm{m}} C$.

WARNING! Although we can compare the hardness using the $\leq^{\mathrm{p}}_{\mathrm{m}}$-reducibility, we should remember that this comparison is w.r.t. the polynomial-time computability and that the comparison finer than the polynomial-time computability cannot be made. For such a comparison, we would need more restrictive reducibilities, such as $\leq^{\log}_{\mathrm{m}}$-reducibility.

Example 4. There are some A and B such that $A \leq^{\mathrm{p}}_{\mathrm{m}} B$ and yet $A \notin \mathrm{DTime}(\ell^{100})$ and $B \in \mathrm{DTime}(\ell^2)$.

In fact, every L in P is $\leq^{\mathrm{p}}_{\mathrm{m}}$-reducible to $\{0\}$ by the following reduction h_L.

$$h_L(x) = \begin{cases} 0, & \text{if } x \in L, \text{ and} \\ 00, & \text{otherwise.} \end{cases}$$

• **Examples of polynomial-time reductions**

Theorem 15. SAT is $\leq_{\mathrm{m}}^{\mathrm{p}}$-reducible to 3SAT. (Since 3SAT $\leq_{\mathrm{m}}^{\mathrm{p}}$ SAT is clear, we have 3SAT $\equiv_{\mathrm{m}}^{\mathrm{p}}$ SAT.)

Proof by Example. Consider the following formula F and explain how to reduce it to an instance G for 3SAT problem.

$$F \;=\; ((X_1 \vee X_2) \equiv (X_2 \oplus (X_1 \wedge \overline{X_3}))).$$

First we define the following formula F' from F.

$$\begin{aligned} F' \;=\; & (U_1 \equiv (X_1 \vee X_2)) \;\wedge\; (U_2 \equiv (X_1 \wedge \overline{X_3})) \\ & \wedge (U_3 \equiv (X_2 \oplus U_2)) \;\wedge\; (U_4 \equiv (U_1 \equiv U_3)) \;\wedge\; U_4. \end{aligned}$$

Though four new variables U_1,..., U_4 are introduced in F', we still have that

$$F \text{ is satisfiable (i.e., } F \in SAT) \iff F' \text{ is satisfiable (i.e., } F' \in SAT).$$

Now we convert each clause of F' into 3CNF formula to make G. For example, $(U_3 \equiv (X_2 \oplus U_2))$ can be converted as follows. (Consider when it becomes false.)

$$\begin{aligned} (U_3 \equiv (X_2 \oplus U_2)) \;=\; & (U_3 \vee X_2 \vee \overline{U_2}) \;\wedge\; (U_3 \vee \overline{X_2} \vee U_2) \\ & \wedge\; (\overline{U_3} \vee X_2 \vee U_2) \;\wedge\; (\overline{U_3} \vee \overline{X_2} \vee \overline{U_2}). \end{aligned}$$

Theorem 16. 3SAT $\leq_{\mathrm{m}}^{\mathrm{p}}$ VC.

Proof by Example. For any 3SAT formula F with n variables and m clauses, we can define some G_F and $k_F = n + 2m$ such that $F \in$ 3SAT $\iff (G_F, k_F) \in$ VC.

To see how to define G_F from F, consider the following 3CNF formula F.

$$F \;=\; (X_1 \vee X_2 \vee \overline{X_3}) \wedge (X_1 \vee \overline{X_2} \vee X_3) \wedge (\overline{X_1} \vee \overline{X_2} \vee X_3).$$

Theorem 17. VC $\leq_{\mathrm{m}}^{\mathrm{p}}$ SAT.

Proof by Idea. Let (G, k) be a given instance for VC problem. We assume that vertices of G are indexed as $1, ..., n$, and let E be the set of edges of G.

For this (G, k), consider the following formula F over $X_1, ..., X_n$.

$$F \;=\; \left(\bigwedge_{(i,j)\in E} (X_i \vee X_j) \right) \;\wedge\; [\, X_1 + X_2 + \cdots + X_n \leq k \,].$$

Then it is easy to see that $(G, k) \in$ VC if and only if F is satisfiable. Thus, the remaining task is to express F as an ordinary Boolean formula.

Note that the addition can be computed by a logical circuit consisting of AND, OR, and NOT gates. Thus, it is intuitively clear that the predicate $\mathit{compare}_k(x_1, ..., x_n) = [\, x_1 + \cdots + x_n \leq k \,]$ can be computed by some circuit C_k.

Though similar, circuits are in general not formulas. But here we can use the previous technique for proving SAT $\leq_{\mathrm{m}}^{\mathrm{p}}$ 3SAT to convert a circuit C to a formula F so that C is satisfiable if and only if F is satisfiable. We use this technique and convert C_k to a formula F_k, thereby converting F to some SAT instance.

By an idea similar to this, we can prove the following key result.

Theorem 18. (Cook '71 and Levin '73) Every $L \in$ NP is $\leq_{\mathrm{m}}^{\mathrm{p}}$-reducible to SAT.

Proof Idea. Let L be any problem in NP, and consider any instance $x \in \{0,1\}^*$ for L. We want to convert it to a Boolean formula F_x so that $x \in L \iff F_x$ is satisfiable.

Recall by definition that there are some polynomial q_L and polynomial-time computable predicate R_L satisfying the NP-condition for L. In particular, the following holds for our x. (For simplifying our argument, we assume that the witness length is exactly $q = q_L(|x|)$.)

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

Again intuitively, we may expect some logical circuit $C_{L,x}$ such that $C_{L,x}(w) = 1$ if and only if $R_L(x, w)$ holds. Note that $C_{L,x}$ has q input gates, each of which receives each bit of a given witness w. Thus, we have $x \in L$ if and only if $C_{L,x}$ is satisfiable (with some inputs $w_1, ..., w_q$). Now we convert this $C_{L,x}$ to a Boolean formula as we did before, which yields a formula F_x such that $x \in L$ if and only if F_x is in SAT.

So far, we have not used the polynomial-time computability of R_L. This is required for constructing F_x in polynomial time. Since R_L is polynomial-time computable, we can show that the circuit $C_{L,x}$ is not so large; it consists of some polynomial number of AND, OR, and NOT gates. This guarantees that F_x can be computed within some polynomial time from x.

• **NP-completeness and completeness notions**

Definition 5. For any complexity class $\mathcal{C}$, a set C is said $\mathcal{C}$*-complete* (or more specifically, $\leq^{\mathrm{p}}_{\mathrm{m}}$*-complete in* $\mathcal{C}$) if
(a) $\forall L \in \mathcal{C}$ [$L \leq^{\mathrm{p}}_{\mathrm{m}} C$], and
(b) C is in $\mathcal{C}$.

Remarks:

1. From Theorem **??** (and the fact that SAT $\in$ NP), we now know that SAT is NP-complete.
2. A problem C satisfying the condition (a) is called $\mathcal{C}$*-hard*. Intuitively, for example, NP-hard problems are as hard as any NP-problem, and NP-complete problems are those that are NP-hard *and* in NP. That is, NP-complete problems are problems that are hardest in NP. (Again remember that the comparison is within the polynomial-time computability.)

Proposition 19. Let C be any NP-complete problem. Then $C \in$ P if and only if P = NP.

Proposition 20. For any A and B, if $A \leq^{\mathrm{p}}_{\mathrm{m}} B$ and A is $\mathcal{C}$-complete, then B is also $\mathcal{C}$-complete.

Thus, for showing NP-completeness of a given problem L, we need to show a reduction from any known NP-complete problem C to L. For example, we have already proved the NP-completeness of 3SAT and VC. The following NP-completeness results are proved in this way.

Theorem 21. HAM, COR, KNAP, SSUM, CLIQUE, IS are all NP-complete.

The NP-completeness has been used to guarantee some sort of hardness to a given NP problem. But for some cases, the completeness notion can be used to prove an actual lower bound for a concrete problem.

Theorem 22. For any problem X, if X is DEXP-complete (or DEXP-hard), then X is not polynomial-time solvable.

Proof. This follows from the fact that P $\subsetneq$ DEXP. Consider any problem L in DEXP$-$P. Then $L \leq^{\mathrm{p}}_{\mathrm{m}} X$, which implies $X \notin$ P because $L \notin$ P.

Example 5. We can show that the attribute grammar circularity test AG-CIRC, a problem of deciding whether a given attribute grammar has a circular definition, is DEXP-complete. Thus, AG-CIRC cannot be solved in polynomial time.

Question?!

Q3. Is there any other application of reducibility and completeness notions?

6. Some Other Complexity Classes

• Relativized complexity classes

In our proof of Theorem **??**, we designed an algorithm for some problem A based on an assumed program for another problem B. Let us generalize this idea. Consider any problem X, and assume that X is solvable *magically* in $O(1)$ time. Then by using this assumed program for X, we design an algorithm for a given problem Y. If some program `IsY` solves Y in polynomial time by using the assumed program for X as a subroutine, then we say that Y is *polynomial-time solvable relative to* X. In other words, `IsY` is regarded as a program solving Y by asking queries to some *oracle* for X, an imaginary mechanism that can answer the question $y \in X$? instantly for any given y.

For any problem X, we define a class P^X to be the set of problems that are polynomial-time solvable relative to X. Classes NP^X, co-NP^X, ... are defined similarly. These classes are called *relativized complexity classes*.

Relativized classes are usually used to discuss important open questions such as $\mathrm{P} \neq \mathrm{NP}$? in relativized worlds. But we can use this mechanism to define new complexity classes. For any complexity class $\mathcal{C}$, $\mathrm{P}^{\mathcal{C}}$ is the set of problems solvable in polynomial time relative to *some* problem in $\mathcal{C}$. We often consider the following complexity classes.

$$\mathrm{P}^{\mathrm{NP}} \subseteq \mathrm{ZPP}^{\mathrm{NP}} \subseteq \mathrm{NP}^{\mathrm{NP}} \subseteq \mathrm{NP}^{\mathrm{NP}^{\mathrm{NP}}} \text{ and } \mathrm{BPP}^{\mathrm{NP}}$$

Remarks:

1. For discussing relativized complexity classes for space classes such as DLOG, NLOG, PSPACE, etc, we usually use a computation model that has a special write-only register for keeping a query to the oracle, and the space for this register is not counted. On the other hand, we usually require that the query length is polynomially bounded.
2. It is easy to see that $\mathrm{P}^{\mathrm{P}} = \mathrm{P}$, $\mathrm{P}^{\text{co-NP}} = \mathrm{P}^{\mathrm{NP}}$, and $\mathrm{NP}^{\text{co-NP}} = \mathrm{NP}^{\mathrm{NP}}$.
3. On the other hand, similar relations $\mathrm{NP} = \mathrm{P}^{\mathrm{NP}}$ and $\mathrm{NP} = \mathrm{NP}^{\mathrm{NP}}$ may not hold. On the contrary, it has been believed that neither of them holds.

• Circuit complexity classes

Logical circuits (circuits in short) are used to investigate the running time of parallel algorithms. They are also used as a combinatorial model of computation.

A *circuit* is a directed acyclic graph. Nodes of the graph, which are called *gates*, are classified into four types. Nodes with no fan-in edge are *input gates*. The other nodes are either AND-, OR-, or NOT-gates. Nodes with no fan-out edge are also called *output gates*. NOT-gates have only one fan-in edge. On the other hand, AND- or OR-gates may have any number of fan-in edge, but unless explicitly specified, we assume that AND- and OR-gates have two fan-in edges.

Remarks:

1. A circuit C with ℓ input gates is used to express computation on input strings of length ℓ. Since each circuit can take care of strings of some fixed length, the whole computation is represented by a family $\{C_\ell\}_{\ell \geq 0}$ of circuits, where each C_ℓ is a circuit with ℓ input gates.
2. In general, we assume no regularity between the circuits in $\{C_\ell\}_{\ell \geq 0}$, and then there may be no way to construct C_ℓ from $\overline{\ell}$. Due to this, a family of circuits is generally called a *nonuniform computation model*.

For the circuit model, we measure computational resource by "depth" and "size". The *depth* of a circuit C, which we denote as depth(C), is the length of the longest path from an input gate to the output gate of C. The *size* of C, size(C), is the total number of AND-, OR-, and NOT-gates.

Now we introduce circuit complexity classes. We start with the following basic complexity classes.

$$\mathrm{SIZE}(s) = \{\ L\ :\ L \text{ is solvable by circuits of size} \leq O(s(\ell))\ \}.$$
$$\mathrm{D\&S}(d, s) = \{\ L\ :\ L \text{ is solvable by circuits of depth} \leq O(d(\ell)) \text{ and of size} \leq O(s(\ell))\ \}.$$

It is intuitively clear that circuit depth corresponds to parallel running time. On the other hand, circuit size corresponds to the combination of the time and space resources of sequential algorithms. Here we introduce the following circuit classes. Roughly speaking, NC is a class of problems solvable in poly. llogarithmic time by some reasonable parallel machine.

$$\mathrm{NC}^k = \bigcup_{q:\mathrm{poly}} \mathrm{D\&S}((\log \ell)^k, q(\ell)), \quad \mathrm{NC} = \bigcup_{k \geq 1} \mathrm{NC}^k.$$

Remarks:
Precisely speaking, in order to have the relation mentioned above between parallel running time and circuit depth, we need to assume the *log-space* uniformity.

Some researchers have been trying to derive nontrivial lower bounds by investigating combinatorial properties of resource bounded computation. For such investigation, we do not need any uniformity, and nonuniform circuit complexity classes, complexity classes defined by nonuniform families of circuits, are enough. The following nonuniform classes have been considered for discussing such combinatorial lower bounds.

$$\mathrm{AC}^0 = \bigcup_{q:\mathrm{poly}} \mathrm{D\&S}(1, q(\ell)), \quad \mathrm{P/poly} = \bigcup_{q:\mathrm{poly}} \mathrm{SIZE}(q(\ell)).$$

Remarks:
For defining AC classes (AC^0 is one of them), we relax the condition on the number of

fan-in edges and allow AND and OR gates to have an arbitrary number of fan-in edges. Otherwise, constant depth circuits would become very weak.

Let us discuss the relation between circuit complexity classes and ordinary complexity classes defined before. The following relation is a key for showing inclusions.

Theorem 23. (Savage '72) $\mathrm{DTime}(t) \cap \mathrm{DSpace}(s) \subseteq \mathrm{D\&S}(t, t \cdot s) \subseteq \mathrm{SIZE}(t \cdot s)$.

Corollary 24. $\mathrm{P} \subseteq \mathrm{P/poly}$.

For low level circuit classes, the following relation holds. (Here we assume the log-space uniformity for all circuit classes.)

Theorem 25. $\mathrm{AC}^0 \subseteq \mathrm{NC}^1 \subseteq \mathrm{DLOG} \subseteq \mathrm{NLOG} \subseteq \mathrm{NC}^2 \subseteq \cdots \subseteq \mathrm{NC} \subseteq \mathrm{P}$.

Theorem 26. (Furst, Saxe, and Sipser '84) $\mathrm{AC}^0 \subsetneq \mathrm{NC}^1$.

• Randomized complexity classes

Recent progress in algorithm and complexity theory has often been made by clever usage of randomness, and randomness is now one of the important tools for designing efficient algorithms. Here we introduce several complexity classes for randomized computation.

Example 6. To see that randomness is an effective tool, consider the following problem.

> Matrix Multiplication Test (MULT-TEST)
> **Instance**: Matrices A, B, and C of size $n \times n$ over some field.
> **Question**: $A \times B \stackrel{?}{=} C$.

For our field, consider $\mathbf{Z}_p$ for some prime number p. Although there is a matrix multiplication algorithm running less than $O(n^3)$ steps, it seems difficult to achieve the above test in $O(n^2)$ steps. On the other hand, randomness gives us the following simple yet efficient algorithm. (Here we simply use n for input size, and count the number of arithmetic operations of $\mathbf{Z}_p$ for running time.)

```
program MTEST(input A, B, C);
   choose r = (r_1,...,r_n) uniformly at random from {0,1,...,p-1}^n;
   u ← A · (B · r);   v ← C · r;
   if u = v then halt(1) else halt(0);
end-program.
```

For a given input A, B, and C, it is easy to see that the algorithm yields 1 if $A \times B = C$. On the other hand, we can show that if $A \times B \neq C$, then with probability $\geq 1/p$, `u` and `v` differs and the algorithm yields 0.

By a similar technique, we can test whether given two polynomials over a finite field are equivalent.

For a randomized computation model, we can use almost the same machinery used for nondeterministic computation. That is, we can describe any randomized algorithm by using the following special if-statement that we introduced for nondeterministic computation.

$$\textbf{if guess} = \texttt{0} \textbf{ then goto } \mathit{label}_1 \textbf{ else goto } \mathit{label}_2$$

But here we give a different interpretation to this statement. When the execution reaches to this statement, we assume that the machine flips a fair coin to determine the value of **guess**. (*Cf.* For nondeterministic computation, we considered that the machine executes two possibilities *in parallel*.)

As before, we call each possible execution a *computation path*. Notice that a computation path is uniquely determined if we fix the values of **guess** used during the execution; furthermore, these choices can be specified as a binary string,i.e., a sequence of 0 and 1. Thus, we often identify one computation path with a binary string that defines this computation path. Intuitively, for executing a program with the above special if-statement, a random binary sequence of enough length is first prepared, and then the program is executed while using each bit of the random sequence when the value of **guess** is needed.

We allow a randomized program to yield different values on different computation path, or even not to halt on some computation path. That is, the output $\mathtt{A}(x)$ of a randomized program $\mathtt{A}$ on input x is a random variable. Since our target is a decision problem, $\mathtt{A}(x)$ is either 0 or 1, but we allow $\mathtt{A}$ yields ? meaning 'don't know'. We say that $\mathtt{A}$ *solves* a decision problem L if $\mathtt{A}(x) = 1 \iff x \in L$ with probability $> 1/2$ for every x. Below we consider only randomized programs that solve some decision problem.

Practically, a randomized program is useless unless its error probability is bounded. Such randomized programs $\mathtt{A}$ are classified into the following three types.

Bounded error: (BP type)

$$\begin{aligned} x \in X &\implies \Pr_{\mathtt{A}}\{\, \mathtt{A}(x) \neq 1 \,\} < \epsilon, \\ x \notin X &\implies \Pr_{\mathtt{A}}\{\, \mathtt{A}(x) \neq 0 \,\} < \epsilon. \end{aligned}$$

One sided error: (R type)

$$\begin{aligned} x \in X &\implies \Pr_{\mathtt{A}}\{\, \mathtt{A}(x) \neq 1 \,\} < \epsilon, \\ x \notin X &\implies \Pr_{\mathtt{A}}\{\, \mathtt{A}(x) \neq 0 \,\} = 0. \end{aligned}$$

One sided error: (co-R type)

$$\begin{aligned} x \in X &\implies \Pr_{\mathtt{A}}\{\, \mathtt{A}(x) \neq 1 \,\} = 0, \\ x \notin X &\implies \Pr_{\mathtt{A}}\{\, \mathtt{A}(x) \neq 0 \,\} < \epsilon, \end{aligned}$$

Zero error: (ZP type)

$$\begin{aligned} x \in X &\implies \Pr_{\mathtt{A}}\{\, \mathtt{A}(x) = 1 \text{ or } ? \,\} = 1, \\ x \notin X &\implies \Pr_{\mathtt{A}}\{\, \mathtt{A}(x) = 0 \text{ or } ? \,\} = 1. \end{aligned}$$

Question?!

Q4. Let A be a randomized program of R type. Which answer of A do you believe?

Remarks:

1. The choice of ϵ is not essential in most cases because one can reduce error probability by a constant factor by executing A a constant number of times.
2. The last condition is called "zero error" because one can always get the correct answer by iterating A's execution until a 1/0 answer is obtained.

Now we define complexity classes in Table 1.

$$
\begin{aligned}
\text{PP} &= \{\, L \;:\; L \text{ is solvable by some poly. time randomized program } \}. \\
\text{BPP} &= \{\, L \;:\; L \text{ is solvable by some poly. time BP type program } \}. \\
\text{RP} &= \{\, L \;:\; L \text{ is solvable by some poly. time R type program } \}. \\
\text{co-RP} &= \{\, L \;:\; L \text{ is solvable by some poly. time co-R type program } \}. \\
\text{ZPP} &= \{\, L \;:\; L \text{ is solvable by some poly. time ZPP type program } \}. \\
\text{RLOG} &= \{\, L \;:\; L \text{ is solvable by some log space RP type program } \}.
\end{aligned}
$$

Proposition 27.
(1) $\text{ZPP} = \text{RP} \cap \text{co-RP} \subseteq \text{RP} \cup \text{co-RP} \subseteq \text{BPP} \subseteq \text{PP}$.
(2) $\text{RP} \subseteq \text{NP}$ and $\text{co-RP} \subseteq \text{co-NP}$.

Theorem 28.
(1) $\text{P} \subseteq \text{ZPP, RP, co-RP, BPP, PP} \subseteq \text{PSPACE}$.
(2) $\text{NP, co-NP} \subseteq \text{PP}$.

Theorem 29. (Adleman '78) $\text{BPP} \subseteq \text{P/poly}$.

Proof Idea. The relation is almost immediate from the following much stronger observation.

Let A be a BP type randomized program whose time complexity is polynomially bounded. Since A halts within polynomial time, we only need, for the random seed for A, a random binary sequence of some polynomial length.

From our intuitive understanding of randomized computation, for each $x \in \{0,1\}^\ell$, there are, say, 2/3 of random binary sequences that are "good", i.e., that have $\mathtt{A}(x)$ to yield the correct answer. Notice that the set of "good" random binary sequences differs depending on x. The key observation is that we can construct a new algorithm AA from A, for which 2/3 of random binary sequences are "good" for *all* input $x \in \{0,1\}^\ell$. That is, if we select a binary sequence randomly, then with high probability, it is *universally good* for AA.

Our circuit is designed for simulating AA with one of these universally good binary sequence that is hard wired (or encoded) in the circuit.