COMPUTATIONAL COMPLEXITY THEORY

Computability is the field of theoretical computer science that formally studies the power of computational models. The objects of the study are mathematical problems, more precisely computations of functions from a certain set of mathematical objects to another such set. A computational model is a specification of the device that performs the computation. A fundamental question in computability theory is whether a given mathematical function has an effective procedure in a given computational model. By *effective* we mean that the procedure finds the correct answer for every instance of the problem. We say that a mathematical problem with an effective procedure is effectively solvable.

An important area of computability theory is computational complexity theory, which is the study of the efficiency of effective procedures. In computational complexity theory we classify problems according to computational resources required by their effective solutions and study relationships among different kinds of efficiency.

THE BIRTH OF THE FIELD

The concept of computation appeared long before the arrival of electronic computers. In fact, Euclid's algorithm for computing the greatest common divisor of integers, one of the oldest known algorithms, is more than two thousand years old. However, it is only recently that we gained formal knowledge of computation and became to learn what we could do and what we could not do with it.

At the end of the nineteenth century, Gottlob Frege (1) conjectured that one should be able to build mathematics from fundamental logic, and formulated a logical system for founding mathematics. Unfortunately, the system soon turned out to yield a contradiction. David Hilbert, a great mathematician of the modern age, strongly supported Frege's idea in his famous 1900 lecture (2). Hilbert believed that there was an effective procedure for deciding in a finite number of mathematical steps whether a given mathematical proposition is true or false. Hilbert and his group put great effort into the search for such a procedure, but they did not make much progress.

In 1931, Kurt Gödel refuted the conjecture by Frege and Hilbert by showing that all elementary proof systems are incomplete in the sense that every such system has statements neither of whose correctness nor incorrectness are provable (3). We refer to this epochal result as Gödel's incompleteness theorem. The focus of this 1931 paper was on the functions of elementary number theory; i.e., the functions from the set of natural numbers to itself. Gödel developed a system for constructing functions of elementary number theory based on certain simple axioms, and studied the class of functions constructible in the system—the class of recursive functions. (Using current terminology this is the class of primitive recursive functions.) Then, Gödel developed a systematic method for encoding a recursive function as a natural number. With this encoding method, not only can one study elementary number theory, but one also can study proof logic of the theory within the theory. He constructed a self-referencing paradox in the theory (like "this statement is false") and showed that one cannot prove or disprove this paradox within it.

Gödel's incompleteness theorem left us to question exactly what kinds of mathematical functions lack effective solutions (i.e., what kinds of mathematical problems are unsolvable) and whether there are different types of unsolvability. In 1936 there emerged papers that shed light on this issue. Alonzo Church introduced lambda calculus and showed that the recursive functions of Gödel are precisely those that are lambda-calculable (4). Stephen Kleene refined Gödel's recursive functions (Kleene proposed the terminology *primitive recursive function*), and presented another proof of the existence of unsolvable problems (5). Alan Turing introduced the Turing machine model (6). Turing presented an analog of Gödel's incompleteness theorem. (His proof, although the statement itself was correct, contained many errors.) Emil Post, independently of Turing, introduced a system similar to the Turing machine model (7).

The papers by Church, Kleene, Post, and Turing demonstrate the significance of Gödel's Incompleteness Theorem, and paved the way to foundations of recursive function theory, lambda calculus theory, and computability theory. Among these models, the Turing machine model is particularly important for computability theory because of its programmable nature. Such capability is exactly what John von Neumann suggested as essential for computers (see Von Neumann Computers). A Turing machine is a device that manipulates an infinitely long one-dimensional tape divided into tape squares. A Turing machine has a head that scans on the tape and moves along the tape one square per computational step. With this head, the machine reads the symbol written in the current tape square and modifies it if necessary.

Surprisingly, it soon became clear that the computational models by Church, Gödel, Kleene, Post, and Turing are all equivalent; every function computable in one system is computable in the other systems, too. This discovery led Church to propose *Church's Thesis* (or *Church-Turing Thesis*), which states that mathematical problems that are intrinsically solvable are those that are effectively solvable in one of (and thus, any of) the three computational models. The thesis also suggests that the notion of effective computability is model-independent.

COMPUTATIONAL COMPLEXITY THEORY

Shortly after the arrival of electronic computers in the mid-twentieth century, it became evident that, while there are problems that are easily solvable on electronic computers, there are some complicated problems, such as calculation of the Ackerman function, that seem to lack quick and easy solutions. This led us to the question, "What are the intrinsic differences between such easy-to-solve and hard-to-solve problems?" Andrej Grzegorczyk [1953] (8), Paul Axt [1959] (9), and Robert Ritchie [1963] (10) studied this question and showed the existence of proper infinite hierarchies in terms of the number of computational steps within the class of recursive functions (and thus, within the class of Turing-computable functions). Hisao Yamada

[1962] brought the discussion of noncomputability to the level of computations that are possible with ordinary computers (11). He studied decision problems that are solvable by Turing machines in such a way that they finish computation about the integer n at the nth step. Yamada called this real-time computation and showed that there are problems that are real-time solvable as well as those that are not. Michael Rabin [1963], independently of Yamada, presented a limitation to real-time computation (12). He showed that with regard to real-time computation, Turing machines that manipulate two separate tapes are more powerful than those with just one tape. These seminal papers suggested that there should exist fine hierarchies in the class of primitive recursive functions with respect to computation time.

In 1965 (a conference presentation was in 1964), Juris Hartmanis and Richard Stearns formulated the concept of time-bounded complexity classes, and proved that the previous expectation was correct (.13). The choice of their model were the multitape Turing machines, those that manipulate some finite number of tapes. The focus of their study was on decision problems, that is, those that ask whether a given word belongs to a language. For a function $T(n)$ from the set of natural numbers to itself, the complexity class with time-bound $T(n)$, denoted TIME($T(n)$), is the collection of all decision problems solvable by Turing machines whose running time is at most $T(n)$ for every n and every input of length n The time complexity classes by Hartmanis and Stearns offer reasonable classifications of decision problems. On one hand, the classifications are not too tight, because Turing machines can speed up their computations by any constant factor, except for some boundary cases (the linear speed-up theorem). On the other hand, they provide proper hierarchies—if one scales up the time-bound by a factor that grows faster than the quadratic, then the class becomes bigger (the time hierarchy theorem). In a subsequent paper, Hartmanis and Stearns, together with Harry Lewis, examined another complexity measure, namely the space-complexity, defined as the number of tape squares that are examined by Turing machines before halting (14). They showed that space-complexity also provides reasonable classifications of problems.

The two papers by Hartmanis, Lewis, and Stearns established foundations of computational complexity theory. Since then researchers have developed numerous concepts as well as proven many important, often striking, observations about feasible computation. Among many contributions of the field, the most influential is perhaps the study of the P = NP? problem (discussed below).

In 1965, Alan Cobham (15) and Jack Edmonds (16) independently proposed to identify tractable mathematical decision problems as those with time complexity of polynomial growth. This is the class that we now call P, the class of problems solvable in polynomial time. Edmonds also discovered that there is a seemingly difficult problem possessing a property that all positive instances have short, easy-to-verify membership certificates while no negative instances have such certificates. The class of problems exhibiting such certificate scheme is what we call NP, the nondeterministic polynomial time. Edmonds then asked whether his problem is tractable, in other words, "Is P = NP?", and this question has been the most important open question in computational complexity theory.

One of the most remarkable achievement in the study of the P = NP? question is the discovery of NP-completeness. In 1971, Stephen Cook introduced the notion of polynomial-time reducibility as a tool for comparing computational complexity of two decision problems (17). A decision problem A is polynomial-time reducible to another decision problem B if there is a polynomial-time Turing algorithm for A that makes use of a hypothetical unit-cost subroutine for B Cook showed that the DNF-tautology problem, the problem of deciding whether a disjunctive-normal-form formula of propositional logic is a tautology, has a special property that every NP-decision problem is polynomial-time reducible to it. An important consequence of this observation is that if the DNF-tautology problem is in P, then every problem in NP has a polynomial-time algorithm, and thus, P = NP. Cook also observed a strong connection of the DNF-tautology problem to NP. Its complementary problem, the DNF-non-tautology problem, is in NP; a short certificate for a nontautological formula is an assignment that falsifies the formula. Hence, polynomial-time decidability of the DNF-tautology is equivalent to P = NP.

In 1972, Richard Karp extended Cook's approach and refined the notion of polynomial-time reducibility (18). In Karp's definition, A is polynomial-time reducible to B if there is a polynomial-time computable transformation f from the instance set of A to that of B such that f maps members of A to those of B and nonmembers of A to those of B Karp established the notion of NP-completeness based on this reducibility. A decision problem C is NP-complete if it is in NP and every decision problem in NP is polynomial-time reducible to C. Karp presented a number of combinatorial NP-complete problems and showed basic techniques for proving NP-completeness of problems. The work by Cook and Karp simplified the P = NP? problem tremendously, because one can discuss the P = NP? problem simply as the question whether a specific NP-complete problem is in P. The discovery of NP-complete problems triggered a gold-rush for more NP-complete problem. In the 1970s alone, researchers identified hundreds of NP-complete problems of practical importance. Currently the list of known NP-completeness consists of thousands of practically important problems in various fields, such as biology, chemistry, combinatorics, logic, number theory, operations research, and VLSI design. These NP-complete problems amount to evidence that NP is perhaps different from P (see Traveling Salesperson problems).

THE TURING MACHINES

Now we formally define Turing machines. A Turing machine carries out computation in steps. The architecture of a Turing machine consists of the tape, the head, and the finite *control* (see Fig. 1). The tape is an infinitely long one-dimensional array of squares, in which each square holds precisely one symbol from the tape alphabet, a finite set of symbols. The head is a device for reading from and writing on tape squares, one at a time, and moves along the tape. The finite control is the unit that determines the action

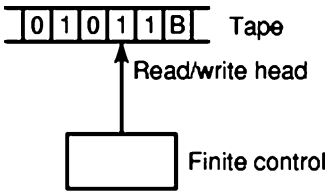

Figure 1. A Turing machine and its three components: tape, read/write head, and finite control.

of a Turing machine. The unit holds a state, an element from the state set. Depending on the current state and the symbol in the tape square of the current head position, the finite control makes the following decisions:

- it may or may not replace the symbol at the head by another symbol;
- it may or may not enter a new state; and
- it may or may not change the position of the head, where if the new position is different from the current one, it has to be one of the two neighboring positions.

At the beginning of computation, a Turing machine holds its input in some consecutive squares of the tape, and all other tape squares hold a special symbol, the blank symbol. Also, the head position is at the first square of the input and the state is a special state called the initial state. Here we demand that the blank symbol should not appear in the input. More precisely, input symbols are from a special set of symbols, the input alphabet, a nonempty subset of the tape alphabet not containing the blank symbol. A Turing machine halts precisely when the state has become one of two special halting states, the accepting state and the rejecting state. The machine accepts the input if it halts by entering the accepting state, and it rejects the input if it halts by entering the rejecting state. The machine could continue its computation indefinitely without reaching a halting state.

A mathematical specification of a Turing machine M is an eight-tuple

$$(\Gamma, B, \Sigma, Q, q_0, q_R, \delta)$$

where Γ is the tape alphabet, $B \in \Gamma$ is the blank symbol, Σ, $\Sigma \neq \emptyset$, and $\Sigma \subseteq \Gamma - \{B\}$, is the input alphabet, Q is the state set, $q_0 \in Q$ is the initial state, $q_A \in Q$ is the accepting state, $q_R \neq q_A \in Q$ is the rejecting state, and δ is the transition function of M, which is a mapping from $Q \times \Gamma$ to $Q \times \Gamma \times \{\leftarrow, \rightarrow, -\}$. The map $\delta(p, a) = (s, b, d)$ specifies that if the machine is currently in the state p and if its head is currently seeing an a, then, in the next step, the machine will be in the state s, a b will replace the a, and the head position will be the current position if $d = -$, the left neighbor of the current one if $d = \leftarrow$, and the right neighbor of the current one if $d = \rightarrow$. For the sake of simplicity we often allow δ to be partial—by leaving out some of its values. We assume that when the state–symbol pair has become one at which δ has no values, the machine immediately enters the rejecting state. Overall, there are three outcomes of computation by a Turing machine; namely, the machine accepts, the machine rejects, and the machine fails to halt.

Now we define decision problems that Turing machines can deal with. A word over an alphabet Σ is a string of finite length all of whose letters are from Σ. A language over Σ is a set of words over Σ.

Definition 1 Let M be a Turing machine. The language that M recognizes, denoted by $L(M)$, is the collection of all words w over M's input alphabet such that M accepts w. The machine M decides $L(M)$ if M halts on all inputs.

An important concept about Turing machines is configuration, which is the status of Turing machines at a specific computational step. More precisely, a configuration consists of the state, the contents of the tape, and the head position. One can view computation by Turing machines as a process of modifying the configuration. For two configurations C and C', if C' results from C by applying the rewriting rule specified by δ, then we describe this fact by notation $C \vdash C'$, and if C' results from applying the rewriting rules specified by δ a finite number of times, then we write $C \vdash^* C'$. We call C an accepting configuration if its state is q_A, and a rejecting configuration if its state is q_F.

Two Examples of Turing Machine Computation

Here we present two examples of Turing machine computation. The first is a machine M, which decides whether a given word x over the alphabet $\{0, 1\}$ is a palindrome, namely, whether the letters of x in the reverse order spell x.

We base the construction of M on three simple properties of palindromes.

1. The empty word is a palindrome.
2. Single letters are palindromes.
3. A word x of length ≥ 2 is a palindrome if and only if the first and the last symbol of x are equal and x without these two symbols is a palindrome.

The states of the machine are the initial state q_0, the accepting state q_A, the rejecting state q_R, and five additional states, s_0, s_1, r_0, r_1, and c. The tape alphabet of M is $\{0, 1, B\}$ and the input alphabet is $\{0, 1\}$. Let u denote the tape contents of M, where the infinitely many blank symbols appearing at the two ends of the tape are omitted. The initial value of u is the input x, and throughout the execution of the algorithm, u will never contain a B.

The machine M executes the following simple loop:

- If $|u|$, the length of u, is at most 1, then accept. Otherwise, compare the two end symbols of u. If they match, then erase them and re-enter the loop; otherwise reject.

More precisely M behaves as follows:

- First M examines the leftmost (or the first) symbol of u. Whenever M does this it is in the state q_0, and in no other occasions, the state is q_0. If the machine sees a B in state q_0, it translates this as u being empty, and accepts. Otherwise, M memorizes the symbol by

entering s_0 if it is a 0 and s_1 if it is a 1, and erases the memorized symbol. Then, it starts moving its head to the right.

- In s_0 as well as in s_1, M looks for the last nonblank symbol of u: M keeps moving its head to the right until it sees a B, then moves the head to the left by one square. While at this, M preserves the symbols it sees and remains in the same state. On accomplishing this task, M changes its state: from s_0 to r_0, and from s_1 to r_1.
- In r_0 as well as in r_1, M compares the current symbol of u and the symbol it has memorized.
  - If they are equal, u was of the form ava with $a \in \{0, 1\}$ at the beginning of the current round of the loop. So, M erases the current symbol, thereby turning u into v, and starts moving the head to the left by entering the state c.
  - If they are not equal but the current symbol is a B, then the head is currently on the square that held the symbol M is memorizing. This implies that u had only one symbol at the beginning of the current round of the loop, so M enters q_A.
  - Otherwise, M enters q_R.
- In state c, M keeps moving the head to the left while preserving the symbols it sees until it encounters a B. When it sees a B, then M moves the head to the right by one square and enters q_0 in order to re-enter the loop.

Table 1. Transitions for Recognizing Palindromes

State / Symbol	0	1	B
q_0	$(s_0, B, \rightarrow)$	$(s_1, B, \rightarrow)$	$(q_A, B, -)$
s_0	$(s_0, 0, \rightarrow)$	$(s_0, 1, \rightarrow)$	$(r_0, B, \leftarrow)$
s_1	$(s_1, 0, \rightarrow)$	$(s_1, 1, \rightarrow)$	$(r_1, B, \leftarrow)$
r_0	$(c, B, \leftarrow)$	$(q_R, B, -)$	$(q_A, B, -)$
r_0	$(q_R, B, -)$	$(c, B, \leftarrow)$	$(q_A, B, -)$
c	$(c, 0, \leftarrow)$	$(c, 1, \leftarrow)$	$(q_0, B, \rightarrow)$

Table 1 shows the exact transition function of M and Fig. 2 illustrates the computation of M on $u = 101$.

The second example is a machine with $\{a, b, c, B\}$ as the tape alphabet that performs the procedure for inserting a c after the first b to the right of the current position. Here, the first B to the right of the starting position acts as the right end of the string. The algorithm of the machine consists of two phases:

- In the first phase, the state is in q_0. In this state, the machine keeps moving the head to the right in search for a b or a B. Upon finding a B, the machine enters the rejecting state q_R, and upon finding a b, it enters q_c and moves the head to the right neighbor of this b.
- In the second phase, the state is one of q_a, q_b, and q_c. These states respectively represent the situations in which the machine is about to insert an a, a b, and a c. The machine executes the insertion by exchanging the symbol in its memory and the symbol it sees; i.e., if the current symbol is x and the current state is q_y, $x, y \in \{a, b, c\}$, then the machine replaces the x by a y and enters q_x. Upon seeing a B, the machine stops its computation by entering q_A.

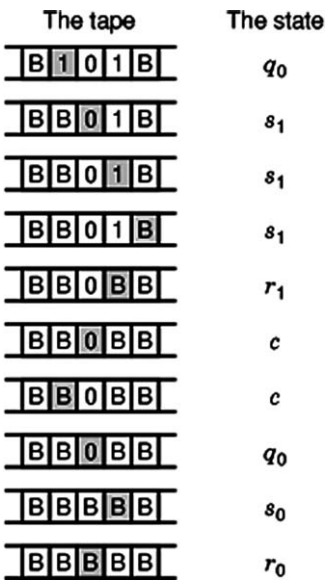

Figure 2. Palindrome recognition. Shaded squares are the locations of the tape. The 1s at the end are matched and erased, then the middle 0 is matched to itself and erased.

Table 2. Transitions for Appending a c after a b

State / Symbol	a	b	c	B
q_0	$(q_0, a, \rightarrow)$	$(r_c, b, \rightarrow)$	$(q_0, c, \rightarrow)$	$(q_R, B, -)$
r_a	$(r_a, a, \rightarrow)$	$(r_b, a, \rightarrow)$	$(r_c, a, \rightarrow)$	$(q_A, B, -)$
r_b	$(r_a, b, \rightarrow)$	$(r_b, b, \rightarrow)$	$(r_c, b, \rightarrow)$	$(q_A, B, -)$
r_c	$(r_a, c, \rightarrow)$	$(r_b, c, \rightarrow)$	$(r_c, c, \rightarrow)$	$(q_A, B, -)$

Table 2 shows the exact transition function of the machine.

Languages That Are Not Turing-Decidable

The difference between recognition by Turing machine and decision by Turing machines is how they treat nonmembers; in the former concept we demand only that the machines avoid accepting nonmembers, while in the latter we demand that they clarify their assertions by rejecting. The incompleteness result that Turing proved in his 1936 paper (6) clarifies that the difference is not trivial.

Theorem 1. Turing's Incompleteness Theorem. There exists a language L that is recognizable by Turing machines but not decidable by them.

For a language L over an alphabet Σ, define $\bar{L}$, the complement of L, to be the set of all words over Σ not belonging to L. It follows immediately from the definition that a language L is decidable by Turing machines if and only if both L and $\bar{L}$ are recognizable by Turing machines. This gives another view of Theorem 1.

Corollary 1 There is a language L that is recognizable by Turing machines such that $\bar{L}$ is not recognizable by Turing machines.

The proof of Theorem 1 consists of the following three observations.

1. One can encode an arbitrary Turing machine, even an arbitrary machine-input pair, as a word over $\{0, 1\}$.
2. There is a universal Turing machine M_{U} that, given an arbitrary machine-input pair (M, x), simulates the computation of M on x.
3. Define L_{N} to be the collection of all encodings w of a Turing machine such that the Turing machine w does not accept the word w. Then no Turing machines cannot recognize L_{N}, let alone decide.

The language L_{N} is a typical undecidable language. There is a rich theory about undecidable problems, founded by Kleene and Post (see Computability). Now we describe sketches of the previous three steps.

Encoding Turing Machines. Suppose we wish to encode a Turing machine M whose tape alphabet, input alphabet, and state set are of size respectively, g, s, and t. Assign numbers from 1 to g to the tape symbols so that those from 1 to s correspond to the input symbols, and so that the number g corresponds to the blank symbol. Assign numbers from 1 to t to the states so that states 1, 2, and 3 are, respectively, the initial state, the accepting state, and the rejecting state. Assign numbers 1, 2, and 3 to the head moves $\leftarrow$, $\rightarrow$, and $-$. With these number assignments, we encode the map $\delta(p, a) = (q, b, d)$ by the word

$$0^{i(p)}10^{i(a)}10^{i(q)}10^{i(b)}10^{i(d)}$$

where $i(x)$ is the index of the object x. Here 1 acts as a delimiter. Concatenating the encodings of all the maps of δ with a longer delimiter 11 gives the description of δ. We encode the sizes with a new delimiter 111

$$0^s1110^s1110^t$$

Appending the encoding of δ to the encoding of the sizes with the delimiter 111 gives the full encoding code(M) of M. Encoding $x = ab \cdots c$ is similar; code(x) is

$$0^{i(a)}10^{i(b)}1 \ldots 10^{i(c)}$$

Encoding of the machine-input pair code(M, x) is code(M)1111code(x).

Universal Turing Machines. An idea for building M_{U} is to split the tape of M_{U} into two parts separated by a delimiter #, where one part holds code(M) and the other holds an encoding of a configuration of M on x. The encoding of a configuration in which the state is q, and in which the tape contents are $a_1a_2 \cdots a_{\mathrm{k}}b_1b_2 \cdots b_{\mathrm{l}}$ with the head residing on the square holding b_1 is

$$0^{i(a_1)}10^{i(a_2)}1 \ldots 10^{i(a_k)}\%0^{i(q)}10^{i(b_1)}10^{i(b_2)}1 \ldots 10^{i(b_l)}$$

where $i(x)$ is the index of the object x and % is another special symbol. The machine M_{U} performs simulation of M on x by constructing the initial configuration, and then simulating each computational step of M on x by rewriting the encoding of a configuration with reference to the first part of the tape. In order to simulate a computational step of M, M_{U} looks for the map to apply. The machine does this by comparing the string $0^{i(q)}10^{i(b_1)}$ with the first two entries of each map in code(M).

The Paradoxical Language. The proof that L_{N} is not recognizable by Turing machines is by way of contradiction. Assume otherwise, and let D be a recognizer of L_{N} and u be the encoding of D. Suppose D accepts u. Then, since D is an acceptor of L_{N}, $u \in L_{\mathrm{N}}$, and thus, D cannot accept u. This is a contradiction. So, D does not accept n. However, if this were the case, then, since D is an acceptor of L_{N}, $u \notin L_{\mathrm{N}}$, and thus, D accepts u, a contradiction. Hence, L_{N} is not recognizable by Turing machines.

Turing Machine Transducers

We formulate the notion of Turing machine transducers by viewing the final contents of the tape as the values they compute. We stipulate that a Turing machine M has an output on an input w if and only if M on w accepts, and we define the output as the shortest string u such that the tape holds at the end of computation the string u in some consecutive tape squares and the blank symbol everywhere else. Then we define the partial function that M computes as a mapping from the words over Σ to those over Γ such that the domain of f is precisely $L(M)$ and such that for all words x over Σ, $f(x)$ is the output of M on input x. A partial function f is Turing-computable if there is a Turing machine M that computes f.

The notion of Turing-computable functions offers another view of Theorem 1. For a partial function f from the words over Σ to those over Γ, define the total-version of f to be the function g defined by: $g(x) = f(x)$ followed by a non-blank symbol $a \in \Gamma$ if $f(x)$ is defined, and the empty word otherwise. Then, Theorem 1 is equivalent to the following.

Corollary 2 There exists a partial function f such that f is Turing-computable but its total-version is not Turing-computable.

COMPUTATIONAL COMPLEXITY THEORY

The computational model of computational complexity theory is the multitape Turing machines, an extension of the Turing machine model. Here Turing machines may have more than one tape to handle. For a natural number $k \geq 1$, a k-tape Turing machine has k separate tapes, on each of which there is a unique read/write head. As in a single-tape Turing machine, the action of a k-tape Turing machine at a single computational step is dependent on its state and the k symbols that its k heads are seeing. More precisely, the transition function is a mapping from $Q \times \Gamma^{\mathrm{k}}$ to $Q \times \Gamma^{\mathrm{k}} \times \{\leftarrow, \rightarrow, -\}^{\mathrm{k}}$. At the beginning of computation the first tape of a k-tape Turing machine has its input in the

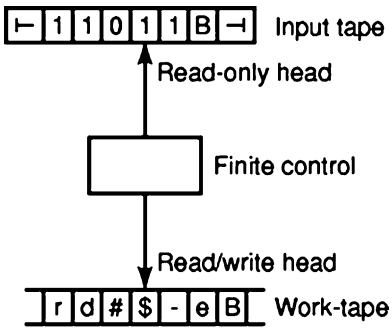

Figure 3. An example of off-line Turing machines. The input tape is read-only. The input is between special symbols, ⊢ and ⊣.

same way as single-tape Turing machines do and the remaining $k-1$ tapes have the blank symbol everywhere. We define the notion of halting, accepting, and rejecting computations as we did for single-tape Turing machines. A special case of the multitape Turing machine model is the off-line Turing machine model, in which the first tape is read-only and has the input between the special end markers ⊢ and ⊣ (see Fig. 3). In contrast on-line Turing machines are those that are not off-line.

Time Complexity Classes

The principal resource of computation is time. For a multitape Turing machine M and an input x to M, the running time of M on x, denoted by $\text{time}_M(x)$, is the number of steps that M expends on x before halting. In the case when M on x does not halt, $\text{time}_M(x)$ is ∞. For a mapping $T(n)$ from the set of natural numbers to itself such that $T(n) \geq n$ for all n, we say that a Turing machine M has the time-bound $T(n)$ if for every input x to M, $\text{time}_M(x) \leq T(|x|)$. The requirement $T(n) \geq n$ comes from our supposition that a reasonable machine should read all its input symbols before halting, and from the fact that reading n symbols requires n steps at least.

A time complexity class is a collection of problems decided by Turing machines sharing the same time bound.

Definition 2 A decision problem L has time complexity $T(n)$ if there is a Turing machine M with time-bound $T(n)$ that decides L. $\text{TIME}(T(n))$ represents the class of all decision problems whose time complexity is $T(n)$.

The Linear Speed-Up Theorem. It is clear from the definition that if $T(n) \leq T'(n)$ for all n, then $\text{TIME}(T(n)) \subseteqq \text{TIME}(T'(n))$. How much larger than $T(n)$ does $T'(n)$ have to be in order for this inclusion to be proper? The linear speed-up theorem (2) states that if $T'(n)$ is within a constant factor of $T(n)$, then the classes are equal.

Theorem 2 *The Linear Speed-Up Theorem*. For every time function $T(n)$ and every constant $c > 0$, $\text{TIME}(T(n)) \subseteqq \text{TIME}(n + c(n + T(n)))$.

The key idea for accelerating computation is to compress many tape symbols into one. Let M be a $T(n)$-time-bounded k-tape Turing machine with Γ as the tape alphabet. We divide the tape of M into blocks of consecutive H tape squares and construct an extension Γ' of Γ. In addition to all the symbols in Γ, Γ' contains symbols for encoding the contents of any size-H block, together with the information whether the head is on any of the H squares, and if so, where. We construct a Turing machine M' that has the same number of tapes as M does, has Γ' as the tape alphabet, and simulates M with the compressed alphabet. Noting that in H steps, the heads of M can touch only two blocks, namely the current block and one of the two neighboring blocks, we can program M' so that it simulates H moves of M in eight steps: four for finding out what the current symbols are, and four for rewriting the compressed encoding so that it reflects the result of applying H moves of M. The input x to M does not respect this compressed encoding scheme. So, the first task of M' is to generate from x in tape 1 the compressed form of x in tape 2. The machine M' simply scans the symbols of x from left to right while erasing them, and accumulates symbols of x; it produces a symbol each time it has accumulated H symbols. After having seen all of x, M' writes any left-over symbols as a single-symbol in Γ' and moves the head to the start of the compressed x in tape 2. This conversion requires $n + [n/H]$ steps. Then, M' carries out the simulation with the role of tape 1 and that of tape 2 exchanged. The total amount of time is

$$T'(n) = n + [n/H] + 8[T(n)/H]$$

So, for any $c > 0$, for a sufficiently large H

$$T'(n) \leq n + c[n + T(n)]$$

for all n.

The following two corollaries yield to the linear speed-up theorem.

Corollary 3 For every $c > 0$ and every time function $T(n)$ such that $\lim_{n\to\infty} T(n)/n = \infty$, $\text{TIME}(T(n)) = \text{TIME}(cT(n))$.

Corollary 4 For every constants c, $\varepsilon > 0$, $\text{TIME}(cn) = \text{TIME}((1+\varepsilon)n)$.

Time Hierarchy Theorems. A function $T(n)$ is time-constructible if there exists a Turing machine C such that $\text{time}_C(x) = T(n)$ for all n and every x of length n. Many functions such as n^k, 2^{kn}, and 2^{n^k} for any natural number k are time-constructible.

Frederick Hennie and Stearns (19) showed that if a time-constructible $T'(n)$ grows more rapidly than $T(n)$ log $T(n)$, then $\text{TIME}(T(n))$ is different from $\text{TIME}(T'(n))$. This significantly improves an early result by Hartmanis and Stearns.

Theorem 3 *The Time Hierarchy Theorem*. If $V(n)$ is time-constructible and $\lim_{n\to\infty} (T(n) \log T(n))/V(n) = 0$, then $\text{TIME}(T(n))$ is a proper subclass of $\text{TIME}(V(n))$.

The proof of this theorem has two parts.

1. Show for every Turing machine M, that there is a two-tape Turing machine simulator S of M such that

for every x

$$\text{time}_s(x) \leq \text{time}_M(x)\log(\text{time}_M(x))$$

2. Based on the first part, construct a language in TIME($V(n)$) not in TIME($T(n)$).

Hennie and Stearns use a very clever idea to prove the first part: the simulator S conducts its simulation not by moving the heads along the tape but by shifting the tape contents so that at the end of a simulation of a single step of M, the work-tape head of S is always at its initial position.

The second part uses a proof technique called diagonalization. Based on the efficient simulation in the first part, we construct a two-tape Turing machine S_U that, on an input w of the form 1^jcode(M) for some integer j and a Turing machine M behaves as follows: S_U spends at most $V(|w|)$ steps for simulating M on code(M) by means of the previous time-efficient simulation with reference to code(M) in the first tape, and accepts if and only if it has discovered that M rejects code(M). Reference to the encoding results in an $O(|\text{code}(M)|)$ multiplicative slowdown in time, but for every $T(n)$-time-bounded M, if h is large enough, then the time given to S_U (i.e., $V(n + |\text{code}(M)|)$) is so large that S_U will finish simulation of M on code(M). So, $L(S_U)$ cannot be decidable by any $T(n)$-time-bounded machine. On the other hand, since S_U is $cV(n)$-time-bounded for some constant $c > 0$, so $L(S_U) \in \text{TIME}(V(n))$. Thus, $\text{TIME}(T(n)) \neq \text{TIME}(V(n))$.

A Gap Theorem. The time-hierarchy theorem demands that the function $T'(n)$ be time-constructible. One may wonder why the constructibility is so important. The reason is that without that notion we would see a very counterintuitive collapse; there would exist a function $f(n)$ such that the time-bounds $f(n)$ and $2^{f(n)}$ would generate the same complexity classes. The result below, which we call a gap theorem, is due to Allan Borodin (1). This is rediscovery of an earlier result by a then-in-Russia mathematician Boris Trakhtenbrot [1967] (20).

Theorem 4 *A Gap Theorem*. There is a total recursive function f such that

$$\text{TIME}(f(n)) = \text{TIME}(2^{f(n)})$$

Space Complexity Classes

Another important resource of computation is space. For an off-line Turing machine M and an input x to M, the space of M on x, denoted by $\text{space}_M(x)$, is the total number of work-tape squares of M that the heads of M have visited at the time of termination. If M on x does not halt, $\text{space}_M(x) = \infty$. For a mapping $S(n)$ from the set of natural numbers to itself such that $S(n) \geq 1$ for all n, we say that a Turing machine M has the space bound $S(n)$ if for every input x to M, $\text{space}_M(x) \leq S(|x|)$. We define space complexity analogously to time complexity, and observe similar basic results.

Definition 3 A decision problem L has space complexity $T(n)$ if there is a Turing machine M with space-bound $S(n)$ that decides L. SPACE($S(n)$) represents the class of all decision problems whose space complexity is $S(n)$.

The following theorem, due to Hartmanis, Lewis, and Stearns (14), is analogous to the linear speed-up theorem (Theorem 2). As in the proof of Theorem 2, the key idea is to compress many symbols into one.

Theorem 5. The Tape Compression Theorem For every space-bound $S(n)$ and every constant $c > 0$, $\text{SPACE}(S(n)) = \text{SPACE}(cS(n))$.

A function $S(n)$ is space-constructible if there exists a Turing machine C halting on all inputs such that $\text{space}_C(x) = S(n)$ for all n and every x of length n. The following theorem, proven in Ref. 14, is analogous to the time hierarchy theorem (Theorem 3).

Theorem 6. The Space Hierarchy Theorem If $S'(n)$ is space-constructible and $\lim_{n\to\infty} S(n)/S'(n) = 0$, then SPACE($S(n)$) is a proper subclass of SPACE($S'(n)$).

Nondeterministic Turing Machines

The nondeterministic Turing machine model is a variation of the Turing machine model in which transition functions may have more than one value. A nondeterministic Turing machine carries out its computation by nondeterministically picking up one move from the list of possible choices, where it enters the rejecting state immediately if there is no choice. We often call Turing machines that are not nondeterministic by deterministic Turing machines. We say that a nondeterministic Turing machine halts on input x if it eventually enters a halting state regardless of its nondeterministic choices. The machine accepts x if it enters the accepting state for some nondeterministic choices; and it rejects x if it enters the rejecting state regardless of its nondeterministic choices. We often view nondeterministic Turing machine computation as a tree (computation tree). The computation tree of a nondeterministic Turing machine N on an input x consists of nodes labeled by configurations of N. The root of the tree is the initial configuration of N on x, and children of a node are the next possible configurations of the node. Here a node is a leaf if and only if it is a halting configuration. So, we call a downward path from the root to a leaf a computation path. The machine N on x accepts if and only if there is a computation path to an accepting configuration. The language that a halting nondeterministic Turing machine N decides, denoted by $L(N)$, is the collection of all w such that N on input w accepts. For a nondeterministic Turing machine N, we define the time-bound and the space-bound of N by taking the maximum over all computation paths of N.

Definition 4 A nondeterministic Turing machine N has time-bound $T(n)$ if for every n, every input x of length n, and every computation path of N on input x, N runs for at most $T(n)$ steps. NTIME($T(n)$) is the class of all languages decided by nondeterministic Turing machines with time-bound $T(n)$.

A nondeterministic Turing machine N has space-bound $S(n)$ if for every n, every input x of length n, and every computation path of N on input x, N on x halts and scans at most $S(n)$ squares of its work-tapes. NSPACE($S(n)$) is the class of all languages decided by nondeterministic Turing machines with space-bound $S(n)$.

The tape compression technique that we used for proving both the linear speed-up theorem and the tape compression theorem allows us to obtain their nondeterministic computation versions. We can also prove a nondeterministic space hierarchy theorem of the same kind: if $S'(n)$, $S(n) \geq \log n$ are space-constructible and $\lim_{n\to\infty} S(n)/S'(n) = 0$, then NSPACE($S(n)$) $\neq$ NSPACE($S'(n)$). The proof makes use of Theorem 10. As to a nondeterministic time hierarchy theorem, the best known result is by Joel Seiferas, Michael Fisher, and Albert Meyer (21): for every $T(n)$ and $T'(n)$, if $\lim_{n\to\infty} T(n+1)/T'(n) = 0$, then NTIME($T'(n)$) properly contains NTIME($T(n)$).

Relationships Among Standard Complexity Classes. It is obvious that TIME($T(n)$) $\subseteqq$ NTIME($T(n)$) for every $T(n)$ and that SPACE($S(n)$) $\subseteqq$ NSPACE($S(n)$) for every $S(n)$ because (deterministic) Turing machines are special nondeterministic Turing machines. Also, for every $k \geq 1$ and t, the number of work-tape squares that a k-work-tape Turing machine can touch in t steps is at most $k \cdot t$. So, using the tape compression theorem, we can prove for all $T(n)$, that TIME($T(n)$) $\subseteqq$ SPACE($T(n)$) and that NTIME($T(n)$) $\subseteqq$ NSPACE($T(n)$). As a matter of fact, even NTIME($T(n)$) $\subseteqq$ SPACE($T(n)$) holds. In order to see this, let N be a $T(n)$-time-bounded nondeterministic Turing machine. Intuitively, we try all computation paths of N on x. For every t, we can write each computation path of length t using t symbols. We increase the value of t from 1 and test whether there is an accepting computation path of length $\leq t$. By the time $t = T(n)$, we will find either that N on x accepts or that all the computation paths are rejecting, therefore N on x does not accept. More precisely, there is a constant H, such that N has at most H possible moves at each step. So, we can view a sequence σ of numbers from 1 to H as a potential computation path of N on x, by letting the ith component of σ represent N's nondeterministic choice at the ith step. By appending at the end of σ dummy entries 0 in the case that the computation lasts for less than t steps, for each $t \geq 1$, we can encode each computation of N until either when t steps have passed or when the computation has stopped, whichever comes first, as a unique element in $S_t = \{0, \ldots, H\}^t$. We can test whether a given sequence is a legitimate encoding by simulating N on x with reference to the sequence. Note that

- N on x accepts if and only if for some $t \geq 1$, some $\sigma \in S_t$ is an accepting computation path of N on x, and that
- N on x rejects if and only if for some $t \geq 1$, all legitimate encodings in S_t are rejecting computation paths of N on x.

The smallest t for which one of the above two conditions holds is at most $T(|x|)$. Thus, we construct a Turing machine S that for $t = 1,2, \cdots$, tests the previous two conditions for t by cycling through the elements σ in S_t and trying to simulate N on x along σ. The space-bound of S is $O(T(|x|))$. So, by the tape compression theorem, we can reduce the bound on space to $T(|x|)$. In summary,

Theorem 7 For all $T(n)$,

$$\mathbf{TIME}(T(n)) \subseteq \mathbf{NTIME}(T(n)) \subseteq \mathbf{SPACE}(T(n)) \subseteq \mathbf{NSPACE}(T(n))$$

The Reachability Problem and Nondeterministic Space Classes. The reachability problem is the problem of deciding, for a given directed graph $G = (V, A)$ and $s, t \in G$, whether there is a directed path from s to t in G. Let N be a halting off-line nondeterministic Turing machine and x be an input N. Then we can state the question whether N on x accepts as an instance of the reachability problem. Suppose N has one work-tape and is $S(n)$-space-bounded, where $S(n) \geq \log n$ is space-constructible. Let V_x be the set of all configurations of N on x. We can assume that the elements of V_x are four tuples (q, i, w, j), where q is the state, i is the head position on the input tape, w is a string of length $S(|x|)$ representing the contents of N's work tape, and j is the head position on w. So, the number of elements $|V_x|$ in V_x satisfies

$$|V_x| = |Q| \times (|x| + 2) \times |\Gamma|^{S(|x|)} \times S(|x|) \leq C^{S(|x|)}$$

for some constant C depending only on N. (The second multiplicative factor is $|x| + 2$ because there are two endmarkers before and after the input.) Define the graph $G_x = (V_x, A_x)$ by drawing an arc from each configuration u to another v if and only if either $u = v$ or $u \vdash v$; i.e., if and only if v results from u by applying at most one computational step of N. Let s_x be the initial configuration of N on x and R_x be the set of all accepting configuration of N on x. Now N on x accepts if and only if there is a path from s_x to some $v \in R_x$ in G_x.

From this observation we obtain a deterministic time-efficient simulation of NSPACE($S(n)$). Let M_x be the adjacency graph of G_x. For each $v \in R_x$, v is reachable from s_x if and only if the (s_x, v)th entry of the $|V_x|$th power of M_x with logical-AND and logical-OR in place of multiplication and addition, respectively, is a 1. Since $S(n)$ is space-constructible, we can construct a Turing machine that decides $L(N)$ by enumerating V_x, constructing M_x, computing the $|V_x|$th power of M_x, then accepting x if and only if the (s_x, v)th entry is a 1 for some $v \in R_x$. We can design the machine so that its running time is at most $(C^{S(n)})^k$ for some constant k depending only on N. Thus,

Theorem 8 If $S(n) \geq \log n$, $S(n)$ is space-constructible, and $L \in$ NSPACE($S(n)$), then there is a constant $D > 0$ such that $L \in$ TIME($D^{S(n)}$).

Another method for solving the reachability problem is by recursion. Suppose we wish to determine whether some t_x is reachable from s_x. Define the predicate $Q(u, v, i)$ as "there is a path from u to v in G_x of length at most 2^i." Note that the length of the path from u_0 to v_0 is at most

$C^{S(|x|)} \leq 2^{cS(|x|)}$ for some integer constant c. So, there is a path from s_0 to t_0 in G_x if and only if $Q(u_0, v_0, cS(|x|))$ evaluates to 1. We can compute $Q(u_0, v_0, cS(|x|))$ with recursion of depth $cS(|x|)$. Note that if $i \geq 1$, then $Q(u, v, i)$ is equivalent to

$$(\exists w)[Q(u, w, i-1) \wedge Q(w, v, i-1)]$$

In our Turing machine algorithm, given u, v, and i, we examine every element in V_x as w, and for each such w, we solve the left subproblem $Q(u, w, i-1)$ first. If this subproblem evaluates to 1, then we move on to the right subproblem $Q(w, v, i-1)$. If this subproblem evaluates to 1, too, then we return with the value 1. Otherwise we continue on to the next w. If we discover that no w makes both parts 1, then we return with the value 0. At the bottom of the recursion, we need to evaluate $Q(y, z, 0)$ for configurations y and z, but this does not require any extra space since we know the machine N. Since the recursion depth is $cS(|x|)$ and each configuration requires $O(S(|x|))$ tape squares, the total storage space of this algorithm is $O(S(n)^2)$ as claimed. This is the result by Walter Savitch (22).

Theorem 9. Savitch's Theorem For every space-constructible $S(n) \geq \log n$, $\text{NSPACE}(S(n)) \subseteq \text{SPACE}(S(n)^2)$.

An important question about nondeterministic classes is the complexity of complementary decision problem. By convention, for a nondeterministic class C, let co-C denote the class of all languages L such that $\bar{L}$ belongs to C. Then, the question we ask is whether $C = \text{co-}C$ for a nondeterministic complexity class C. Our intuition tells that equality is unlikely to hold, because the way nondeterministic Turing machines act on members is very different than the way they act on nonmembers; that is, they have accepting paths for members while they do not for nonmembers. However, to our great astonishment, Neil Immerman (23) and Robert Szelepcsényi (24) independently showed in the late 1980s that such intuition was wrong for nondeterministic space complexity classes.

Theorem 10. The Immerman-Szelepcsényi Theorem For every space-constructible function $S(n) \geq \log n$, $\text{NSPACE}(S(n)) = \text{co-NSPACE}(S(n))$.

Broader Classifications of Problems. In computational complexity we often use broader classifications of decision problems than those discussed. In the broader classifications, the standard classes are L, NL P, NP, and PSPACE. Among these the smallest class is L (the deterministic logspace), which is $\text{SPACE}(\log n)$. A representative is the problem of computing a specified bit of the product of two binary numbers. The class NL (the nondeterministic deterministic logspace) is $\text{NSPACE}(\log n)$. Representatives of this class are the reachability problem and the maze-threadability problem. The class P (the polynomial time) is $\cup_{k \geq 0} \text{TIME}(n^k)$. A standard complete problem in this class is the circuit value problem, a problem of computing outputs of logic-gate circuits. The class NP (the nondeterministic polynomial time) is $\cup_{k \geq 0} \text{NTIME}(n^k)$. Representatives of this class are NP-complete problems. To name a few, standard NP-complete problems are the clique problem (the problem of testing whether a given undirected graph contains a complete graph of given size), the Hamilton path problem (the problem of deciding whether a given directed graph contains a node disjoint cycle that visits all the nodes), and the satisfiability problem (the problem of deciding whether a given formula of propositional logic has satisfying assignment). A related important class is co-NP, the complementary class of NP. The DNF-tautology problem is complete for this class. The class PSPACE (the polynomial space) is $\cup_{k \geq 0} \text{SPACE}(n^k)$. A typical complete problem for this class is the game GO (the problem of deciding whether there exists a winning strategy for the first player starting from the current placement of the pebbles). These classes become larger in the order of their mention; i.e.,

$$\text{L} \subseteq \text{NL} \subseteq \text{P} \subseteq \text{NP} \subseteq \text{PSPACE}$$

The only proper inequalities we know are: NL $\neq$ PSPACE (by combining Savitch's theorem and the space hierarchy theorem) and L $\neq$ PSPACE, which results from NL $\neq$ PSPACE. Defining NPSPACE as $\cup_{k \geq 0} \text{NSPACE}(n^k)$ is possible, but by Savitch's theorem, NPSPACE = co-NPSPACE = PSPACE, so defining the class does not make sense. Also, by the Immerman–Szelepcsényi Theorem, co-NL = NL. We still do not know whether co-NP = NP. Nor do we know whether the intersection of NP and co-NP properly contains P. An important problem in this intersection but not known to be in P was Primality, the problem of testing whether a given binary integer is a prime number. Recently, Manindra Agrawal et al., (25) proved that this problem is polynomial-time solvable, and thus in P, resolving a long-time open question.

BIBLIOGRAPHY

1. G. Frege, Über die Begriffsschrift des Herrn Peano und meine eigene, *Berichte über die Verhandlungen der Königlich Sächsischen Gesellschaft der Wissenschaftern zu Leipzig, Mathematisch-physikalische Klasse*, **48**: 361–378, 1896.
2. D. Hilbert, Mathematical problems, *Bull. Amer. Math. Soc.*, **8**: 161–190, 1901.
3. K. Gödel, Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme, I, *Monatshefte für Mathematik und Physik* **38**: 173–198, 1931.
4. A. Church, An unsolvable problem of elementary number theory, *Amer. J. Math.*, **58**: 345–363, 1936.
5. S. C. Kleene, General recursive functions of natural numbers, *Mathematische Annalen*, **112**: 727–742, 1936.
6. A. M. Turing, On computable numbers with an application to the Entscheidungsproblem, *Proc. London Math. Soc.*, **2** (42): 230–265, 1936. A correction, *ibid.*,**2** (43): 544–546.
7. E. L. Post, Finite combinatory processes-formulation, I, *J. Symbol. Logic*, **1**: 103–105, 1936.
8. A. Grzegorczyk, Some classes of recursive functions, *Rosprawy Matematyczne*, **4**, Warsaw, Poland: Instytut Matematyczne Polskiej Akademie Nauk, 1953.
9. P. Axt, On a subrecursive hierarchy and primitive recursive degrees, *Trans. Amer. Math. Soc.*, **92**: 85–105, 1959.
10. R. W. Ritchie, Classes of predictably computable functions, *Trans. Amer. Math. Soc.*, **106**: 139–173, 1963.

11. H. Yamada, Real-time computation and recursive functions not real-time computable, *IEEE Trans. Elec. Comput.*, **11**: 753–760, 1962.
12. M. O. Rabin, Real-time computation, *Israel J. Math.* **1** (4): 203–211, 1963.
13. J. Hartmanis, R. E. Stearns, On the computational complexity of algorithms. *Trans. Amer. Math. Soc.*, **117**: 285–306, 1965.
14. R. E. Stearns, J. Hartmanis, P. M. Lewis II, Hierarchies of memory limited computation, *Conf. Record Symp. Switching Circuit Theory Logic Des*., pp. 179–190, New York: IEEE Computer Group, 1965.
15. A. Cobham, The intrinsic computational difficulty of functions, Proc. 1964 Congr. Logic, Math. Phil. Sci., Amsterdam: North Holland, 1964, pp. 24–30.
16. J. Edmonds, Paths, trees, and flowers, *Canad. J. Math.* **17**: 449–467, 1965.
17. S. A. Cook, The complexity of theorem proving procedures, Proc. 3rd Annu. Symp. Theory Comput., New York: ACM Press, 1971, pp. 151–158.
18. R. M. Karp, Reducibility among combinatorial problems, in R. Miller and J. Thatcher (eds.), *Complexity of Computer Computations*, New York: Plenum Press, 1972, pp. 85–103.
19. F. C. Hennie, R. E. Stearns, Two-tape simulation of multitape Turing machines, *J. Assoc. Comput. Mach.*, **13**: 533–546, 1966.
20. B. A. Trakhtenbrot, Turing computations with logarithmic delay, *Algebra i Logika*, **3** (4): 33–48, 1964.
21. J. I. Seiferas, M. J. Fisher, A. R. Meyer, Separating nondeterministic time complexity classes, *J. Assoc. Comput. Mach.*, **25**: 146–167, 1975.
22. W. J. Savitch, Relationships between nondeterministic and deterministic tape complexities, *J. Comp. Syst. Sci.*, **4**: 177–192, 1970.
23. N. Immerman, Nondeterministic space is closed under complement, *SIAM J. Comput.*, **17**: 267–276, 1989.
24. R. Szelepcsényi, The method of forced enumeration for nondeterministic automata, *Acta Inf.*, **26**: 279–284, 1988.
25. M. Agwaral, N. Kayal, and M. Saxena, PRIMES in P, *Ann. Math*., **160**: 281–793, 2004.

BIBLIOGRAPHY

Reading List

S. A. Cook, The complexity of theorem proving procedures. In *Proc. Third Annu. Symp. Theory Comput*., pp. 151–158, New York, NY: ACM Press, 1971. One of the standard papers on NP-completeness (see also Karp 1972).

M. D. Davis, (ed.), *The Undecidable: Basic Papers on Undecidable Propositions, Unsolvable Problems, and Computable Functions*, Hewlett, NY: Raven Press, 1965. Reprints of fundamental work by Gödel, Church, Kleene, Post, and Turing.

M. D. Davis, R. Sigal, E. J. Weyuker, *Computability, Complexity, and Languages*, 2nd. ed., San Diego, CA: Academic Press, 1994. Standard textbook about Turing-machine based computability theory.

M. A. Garey, D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, New York: W. H. Freeman, 1979. An enjoyable book about NP-completeness. The book provides a list of hundreds of then-known NP-complete problems.

J. Hartmanis, Observations about the development of theoretical computer science. In Proc. Twentieth Annu. Conf. Found. Comput. Sci., pp. 224–233, New York: IEEE Computer Society Press, 1979. Enjoyable reading about early history of the field of computational complexity theory.

J. Hartmanis, On computational complexity and the nature of computer science, *Commun. Assoc. Comput. Mach.*, **37** (10): 37–43, 1994. Enjoyable reading about early history of the field of computational complexity theory.

J. Hartmanis, R. E. Stearns, On the computational complexity of algorithms, *Trans. Amer. Math. Soc.*, **117**: 285–306, 1965. Epochal work on computational complexity theory (see also Stearns et al. 1965).

J. Heijenoort,(ed.) *Frege and Gödel: Two Fundamental Texts in Mathematical Logic*, Cambridge, MA: Harvard University Press, 1965. English translation of Frege's work.

J. E. Hopcroft, J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation*, Reading, MA: Addison-Wesley, 1979. Standard textbook about Turing-machine based computability theory.

D. S. Johnson, A catalog of complexity classes. In J. van Leeuwen (ed.), *Handbook of Theoretical Computer Science*, Volume A: Algorithm and Complexity, pp. 67–161, Cambridge, MA: The MIT Press, 1990. Provides a catalog of complexity classes.

R. M. Karp, Reducibility among combinatorial problems. In R. Miller and J. Thatcher (ed.), *Complexity of Computer Computations*, pp. 85–103, New York: Plenum Press, 1972. One of the standard papers on NP-completeness (see also Cook, 1971).

K. Kobayashi, On proving time constructibility of functions, *Theoret. Comp. Sci.*, **35**: 215–225, 1985. An extensive study of time-constructible functions.

C. H. Papadimitriou, *Computational Complexity*, Reading, MA: Addison-Wesley, 1994. Covers current trends in computational complexity theory.

H. Rogers, Jr., *Theory of Recursive Functions and Effective Computability*, New York: McGraw-Hill, 1967. Standard textbook about recursive functions and degrees of unsolvability.

R. E. Stearns, J. Hartmanis, P. M. Lewis II, Hierarchies of memory limited computation. In *Conf. Record Symp. Switching Circuit Theory and Logic. Design*, pp. 179–190, New York: IEEE Computer Group, 1965. Epochal work on computational complexity theory (see also Hartmanis and Stearns 1965).

MITSUNORI OGIHARA
University of Rochester,
Rochester, NY