Computational Complexity: A Modern Approach

Draft of a book: Dated January 2007
Comments welcome!

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

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

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

Chapter 1

The computational model —and why it doesn't matter

> *"The idea behind digital computers may be explained by saying that these machines are intended to carry out any operations which could be done by a human computer. The human computer is supposed to be following fixed rules; he has no authority to deviate from them in any detail. We may suppose that these rules are supplied in a book, which is altered whenever he is put on to a new job. He has also an unlimited supply of paper on which he does his calculations."*
> Alan Turing, 1950

> *"[Turing] has for the first time succeeded in giving an absolute definition of an interesting epistemological notion, i.e., one not depending on the formalism chosen."*
> Kurt Gödel, 1946

The previous chapter gave an informal introduction to computation and *efficient computations* in context of arithmetic. IN this chapter we show a more rigorous and general definition. As mentioned earlier, one of the surprising discoveries of the 1930s was that all known computational models are able to *simulate* each other. Thus the set of *computable* problems does not depend upon the computational model.

In this book we are interested in issues of *computational efficiency*, and therefore in classes of "efficiently computable" problems. Here, at first glance, it seems that we have to be very careful about our choice of a computational model, since even a kid knows that whether or not a new video game program is "efficiently computable" depends upon his computer's hardware. Surprisingly though, we can restrict attention to a single abstract computational model for studying many questions about efficiency—the Turing machine. The reason is that the Turing Machine seems able to *simulate* all physically realizable computational models with very little loss of efficiency. Thus the set of "efficiently computable" problems is at least as large for the Turing Machine as for any other model. (One possible exception is the quantum computer model, but we do not currently know if it is physically realizable.)

The *Turing machine* is a simple embodiment of the age-old intuition that computation consists of applying mechanical rules to manipulate numbers, where the person/machine doing the manipulation is allowed a *scratch pad* on which to write the intermediate results. The Turing Machine can be also viewed as the equivalent of any modern programming language — albeit one with no built-in prohibition about memory size[1]. In fact, this intuitive understanding of computation will suffice for most of the book and most readers can skip many details of the model on a first reading, returning to them later as needed.

The rest of the chapter formally defines the Turing Machine and the notion of *running time*, which is one measure of computational effort. It also presents the important notion of the *universal Turing machine*. Section 1.5 introduces a class of "efficiently computable" problems called **P** (which stands for *Polynomial* time) and discuss its philosophical significance. The section also points out how throughout the book the definition of the Turing Machine and the class **P** will be a starting point for definitions of many other models, including nondeterministic, probabilistic and quantum Turing machines, Boolean circuits, parallel computers, decision trees, and communication games. Some of these models are introduced to study arguably realizable modes of physical computation, while others are mainly used to gain insights on Turing machines.

1.1 Encodings and Languages: Some conventions

Below we specify some of the notations and conventions used throughout this chapter and this book to represent computational problem. We make use of some notions from discrete math such as strings, sets, functions, tuples, and graphs. All of these notions are reviewed in Appendix **??**.

1.1.1 Representing objects as strings

In general we study the complexity of computing a function whose input and output are finite strings of bits. (A string of bits is a finite sequence of zeroes and ones. The set of all strings of length n is denoted by $\{0,1\}^n$, while the set of all strings is denoted by $\{0,1\}^* = \cup_{n\geq 0}\{0,1\}^n$; see Appendix A.) Note that simple encodings can be used to represent general objects—integers, pairs of integers, graphs, vectors, matrices, etc.— as strings of bits. For example, we can represent an integer as a string using the binary expansion (e.g., 34 is represented as 100010) and a graph as its adjacency matrix (i.e., an n vertex graph G is represented by an $n \times n$ 0/1-valued matrix A such that $A_{i,j} = 1$ iff the edge (i,j) is present in G). We will typically avoid dealing explicitly with such low level issues of representation, and will use $\llcorner x \lrcorner$ to denote some canonical (and unspecified) binary representation of the object x. Often we will drop the symbols $\llcorner \lrcorner$ and simply use x to denote both the object and its representation.

Representing pairs and tuples. We use the notation $\langle x, y\rangle$ to denote the ordered pair consisting of x and y. A canonical representation for $\langle x, y\rangle$ can be easily obtained from the representations of x and y. For example, we can first encode $\langle x, y\rangle$ as the string $\llcorner x \lrcorner \circ \# \circ \llcorner y \lrcorner$ over the alphabet

[1] Though the assumption of an infinite memory may seem unrealistic at first, in the complexity setting it is of no consequence since we will restrict the machine to use a finite amount of tape cells for any given input (the number allowed will depend upon the input size).

$\{0, 1, \#\}$ (where $\circ$ denotes concatenation) and then use the mapping $0 \mapsto 00, 1 \mapsto 11, \# \mapsto 01$ to convert this into a string of bits. To reduce notational clutter, instead of $\llcorner\langle x, y\rangle\lrcorner$ we use $\langle x, y\rangle$ to denote not only the pair consisting of x and y but also the representation of this pair as a binary string. Similarly, we use $\langle x, y, z\rangle$ to denote both the ordered triple consisting of x, y, z and its representation, and use similar notation for 4-tuples, 5-tuples etc..

1.1.2 Decision problems / languages

An important special case of functions mapping strings to strings is the case of *Boolean* functions, whose output is a single bit. We identify such a function f with the set $L_f = \{x : f(x) = 1\}$ and call such sets *languages* or *decision problems* (we use these terms interchangeably). We identify the computational problem of computing f (i.e., given x compute $f(x)$) with the problem of deciding the language L_f (i.e., given x, decide whether $x \in L_f$).

Example 1.1
By representing the possible invitees to a dinner party with the vertices of a graph having an edge between any two people that can't stand one another, the dinner party computational problem from the introduction becomes the problem of finding a maximum sized *independent set* (set of vertices not containing any edges) in a given graph. The corresponding language is:

$$\mathsf{INDSET} = \{\langle G, k\rangle : \exists S \subseteq V(G) \text{ s.t. } |S| \geq k \text{ and } \forall u, v \in S, \overline{u\,v} \notin E(G)\}$$

An algorithm to solve this language will tell us, on input a graph G and a number k, whether there exists a conflict-free set of invitees, called an *independent set*, of size at least k. It is not immediately clear that such an algorithm can be used to actually find such a set, but we will see this is the case in Chapter 2. For now, let's take it on faith that this is a good formalization of this problem.

1.1.3 Big-Oh notation

As mentioned above, we will typically measure the computational efficiency algorithm as the number of a basic operations it performs as *a function of its input length*. That is, the efficiency of an algorithm can be captured by a function T from the set of natural numbers $\mathbb{N}$ to itself such that $T(n)$ is equal to the maximum number of basic operations that the algorithm performs on inputs of length n. However, this function is sometimes be overly dependant on the low-level details of our definition of a basic operation. For example, the addition algorithm will take about three times more operations if it uses addition of single digit *binary* (i.e., base 2) numbers as a basic operation, as opposed to *decimal* (i.e., base 10) numbers. To help us ignore these low level details and focus on the big picture, the following well known notation is very useful:

DEFINITION 1.2 (BIG-OH NOTATION)
If f, g are two functions from $\mathbb{N}$ to $\mathbb{N}$, then we **(1)** say that $f = O(g)$ if there exists a constant c such that $f(n) \leq c \cdot g(n)$ for every sufficiently large n, **(2)** say that $f = \Omega(g)$ if $g = O(f)$, **(3)** say that $f = \Theta(g)$ is $f = O(g)$ and $g = O(f)$, **(4)** say that $f = o(g)$ if for every $\epsilon > 0$, $f(n) \leq \epsilon \cdot g(n)$ for every sufficiently large n, and **(5)** say that $f = \omega(g)$ if $g = o(f)$.

To emphasize the input parameter, we often write $f(n) = O(g(n))$ instead of $f = O(g)$, and use similar notation for $o, \Omega, \omega, \Theta$.

EXAMPLE 1.3
Here are some examples for use of big-Oh notation:

1. If $f(n) = 100n \log n$ and $g(n) = n^2$ then we have the relations $f = O(g)$, $g = \Omega(f)$, $f = o(g)$, $g = \omega(f)$.

2. If $f(n) = 100n^2 + 24n + 2logn$ and $g(n) = n^2$ then $f = O(g)$. We will often write this relation as $f(n) = O(n^2)$. Note that we also have the relation $g = O(f)$ and hence $f = \Theta(g)$ and $g = \Theta(f)$.

3. If $f(n) = \min\{n, 10^6\}$ and $g(n) = 1$ for every n then $f = O(g)$. We use the notation $f = O(1)$ to denote this. Similarly, if h is a function that tends to infinity with n (i.e., for every c it holds that $h(n) > c$ for n sufficiently large) then we write $h = \omega(1)$.

4. If $f(n) = 2^n$ then for every number $c \in \mathbb{N}$, if $g(n) = n^c$ then $g = o(f)$. We sometimes write this as $2^n = n^{\omega(1)}$. Similarly, we also write $h(n) = n^{O(1)}$ to denote the fact that h is bounded from above by some polynomial. That is, there exist a number $c > 0$ such that for sufficiently large n, $h(n) \leq n^c$.

For more examples and explanations, see any undergraduate algorithms text such as [**?**, **?**] or Section 7.1 in Sipser's book [**?**].

1.2 Modeling computation and efficiency

We start with an informal description of computation. Let f be a function that takes a string of bits (i.e., a member of the set $\{0,1\}^*$) and outputs, say, either 0 or 1. Informally speaking, an *algorithm* for computing f is a set of mechanical rules, such that by following them we can compute $f(x)$ given any input $x \in \{0,1\}^*$. The set of rules being followed is fixed (i.e., the same rules must work for all possible inputs) though each rule in this set may be applied arbitrarily many times. Each rule involves one or more of the following "elementary" operations:

1. Read a bit of the input.

2. Read a bit (or possibly a symbol from a slightly larger alphabet, say a digit in the set $\{0, \ldots, 9\}$) from the "scratch pad" or working space we allow the algorithm to use.

   Based on the values read,

3. Write a bit/symbol to the scratch pad.

4. Either stop and output 0 or 1, or choose a new rule from the set that will be applied next.

Finally, the *running time* is the number of these basic operations performed.
Below, we formalize all of these notions.

1.2.1 The Turing Machine

The *k-tape Turing machine* is a concrete realization of the above informal notion, as follows (see Figure 1.1).

Scratch Pad: The scratch pad consists of k tapes. A *tape* is an infinite one-directional line of cells, each of which can hold a symbol from a finite set Γ called the *alphabet* of the machine. Each tape is equipped with a *tape head* that can potentially read or write symbols to the tape one cell at a time. The machine's computation is divided into discrete time steps, and the head can move left or right one cell in each step.

The first tape of the machine is designated as the *input* tape. The machine's head can can only read symbols from that tape, not write them —a so-called read-only head.

The $k-1$ read-write tapes are called *work tapes* and the last one of them is designated as the *output tape* of the machine, on which it writes its final answer before halting its computation.

Finite set of operations/rules: The machine has a finite set of *states*, denoted Q. The machine contains a "register" that can hold a single element of Q; this is the "state" of the machine at that instant. This state determines its action at the next computational step, which consists of the following: **(1)** read the symbols in the cells directly under the k heads **(2)** for the $k-1$ read/write tapes replace each symbol with a new symbol (it has the option of not changing the tape by writing down the old symbol again), **(3)** change its register to contain another state from the finite set Q (it has the option not to change its state by choosing the old state again) and **(4)** move each head one cell to the left or to the right.

One can think of the Turing machine as a simplified modern computer, with the machine's tape corresponding to a computer's memory, and the transition function and register corresponding to the computer's central processing unit (CPU). However, it's best to think of Turing machines as simply a formal way to describe algorithms. Even though algorithms are often best described by plain English text, it is sometimes useful to express them by such a formalism in order to argue about them mathematically. (Similarly, one needs to express an algorithm in a programming language in order to execute it on a computer.)

Formal definition. Formally, a TM M is described by a tuple (Γ, Q, δ) containing:

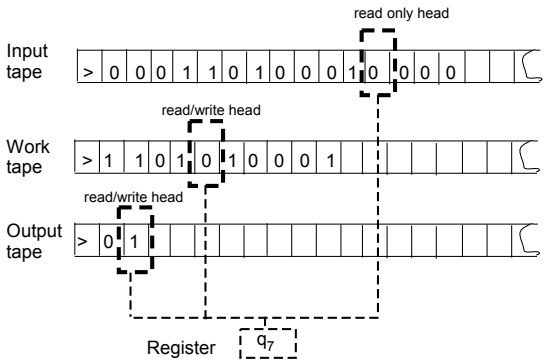

Figure 1.1: A snapshot of the execution of a 3-tape Turing machine M with an input tape, a work tape, and an output tape.

- A set Γ of the symbols that M's tapes can contain. We assume that Γ contains a designated "blank" symbol, denoted $\square$, a designated "start" symbol, denoted $\triangleright$ and the numbers 0 and 1. We call Γ the *alphabet* of M.
- A set Q of possible states M's register can be in. We assume that Q contains a designated start state, denoted q_{start} and a designated halting state, denoted q_{halt}.
- A function $\delta : Q \times \Gamma^k \to Q \times \Gamma^{k-1} \times \{\mathsf{L}, \mathsf{S}, \mathsf{R}\}^k$ describing the rule M uses in performing each step. This function is called the *transition function* of M (see Figure 1.2.)

IF			THEN			
input symbol read	work/ output tape symbol read	current state	move input head	new work/ output tape symbol	move work/ output tape	new state
⋮	⋮	⋮	⋮	⋮	⋮	⋮
a	b	q	→	b'	←	q'
⋮	⋮	⋮	⋮	⋮	⋮	⋮

Figure 1.2: The transition function of a two tape TM (i.e., a TM with one input tape and one work/output tape).

If the machine is in state $q \in Q$ and $(\sigma_1, \sigma_2, \ldots, \sigma_k)$ are the symbols currently being read in the k tapes, and $\delta(q, (\sigma_1, \ldots, \sigma_{k+1})) = (q', (\sigma'_2, \ldots, \sigma'_k), z)$ where $z \in \{\mathsf{L}, \mathsf{SR}\}^k$ then at the next step the σ symbols in the last $k-1$ tapes will be replaced by the σ' symbols, the machine will be in state

q', and the $k+1$ heads will move Left, Right or Stay in place, as given by z. (If the machine tries to move left from the leftmost position of a tape then it will stay in place.)

All tapes except for the input are initialized in their first location to the *start* symbol $\triangleright$ and in all other locations to the *blank* symbol $\square$. The input tape contains initially the start symbol $\triangleright$, a finite non-blank string ("the input"), and the rest of its cells are initialized with the blank symbol $\square$. All heads start at the left ends of the tapes and the machine is in the special starting state q_{start}. This is called the *start configuration* of M on input x. Each step of the computation is performed by applying the function δ as described above. The special halting state q_{halt} has the property that once the machine is in q_{halt}, the transition function δ does not allow it to further modify the tape or change states. Clearly, if the machine enters q_{halt} then it has *halted*. In complexity theory we are typically only interested in machines that halt for every input in a finite number of steps.

Now we formalize the notion of running time. As every non-trivial algorithm needs to at least read its entire input, by "quickly" we mean that the number of basic steps we use is small *when considered as a function of the input length*.

Definition 1.4 (Computing a function and running time)
Let $f : \{0,1\}^* \to \{0,1\}^*$ and let $T : \mathbb{N} \to \mathbb{N}$ be some functions, and let M be a Turing machine. We say that M *computes* f *in* $T(n)$*-time*[2] if for every $x \in \{0,1\}^*$, if M is initialized to the start configuration on input x, then after at most $T(|x|)$ steps it halts with $f(x)$ written on its output tape.
We say that M *computes* f if it computes f in $T(n)$ time for some function $T : \mathbb{N} \to \mathbb{N}$.

Remark 1.5 (Time-constructible functions)
We say that a function $T : \mathbb{N} \to \mathbb{N}$ is *time constructible* if $T(n) \geq n$ and there is a TM M that computes the function $x \mapsto \llcorner T(|x|) \lrcorner$ in time $T(n)$. (As usual, $\llcorner T(|x|) \lrcorner$ denotes the binary representation of the number $T(|x|)$.)

Examples for time-constructible functions are n, $n \log n$, n^2, 2^n. Almost all functions encountered in this book will be time-constructible and, to avoid annoying anomalities, we will restrict our attention to time bounds of this form. (The restriction $T(n) \geq n$ is to allow the algorithm time to read its input.)

Example 1.6
Let PAL be the Boolean function defined as follows: for every $x \in \{0,1\}^*$, $\mathsf{PAL}(x)$ is equal to 1 if x is a *palindrome* and equal to 0 otherwise. That is, $\mathsf{PAL}(x) = 1$ if and only if x reads the same from left to right as from right to left (i.e., $x_1 x_2 \ldots x_n = x_n x_{n-1} \ldots x_1$). We now show a TM M that computes PAL within less than $3n$ steps.

[2] Formally we should write "T-time" instead of "$T(n)$-time", but we follow the convention of writing $T(n)$ to emphasize that T is applied to the input length.

Our TM M will use 3 tapes (input, work and output) and the alphabet $\{\triangleright, \square, 0, 1\}$. It operates as follows:

1. Copy the input to the read/write work tape.
2. Move the input head to the beginning of the input.
3. Move the input-tape head to the right while moving the work-tape head to the left. If at any moment the machine observes two different values, it halts and output 0.
4. Halt and output 1.

We now describe the machine more formally: The TM M uses 5 states denoted by $\{q_{\mathsf{start}}, q_{\mathsf{copy}}, q_{\mathsf{right}}, q_{\mathsf{test}}, q_{\mathsf{halt}}\}$. Its transition function is defined as follows:

1. On state q_{start}, move the input-tape head to the right, and move the work-tape head to the right while writing the start symbol $\triangleright$; change the state to q_{copy}. (Unless we mention this explicitly, the function does not change the output tape's contents or head position.)
2. On state q_{copy}:
   - If the symbol read from the input tape is not the blank symbol $\square$ then move both the input-tape and work-tape heads to the right, writing the symbol from the input-tape on the work-tape; stay in the state q_{copy}.
   - If the symbol read from the input tape is the blank symbol $\square$, then move the input-tape head to the left, while keeping the work-tape head in the same place (and not writing anything); change the state to q_{right}.
3. On state q_{right}:
   - If the symbol read from the input tape is not the start symbol $\triangleright$ then move the input-head to the left, keeping the work-tape head in the same place (and not writing anything); stay in the state q_{right}.
   - If the symbol read from the input tape is the start symbol $\triangleright$ then move the input-tape to the right and the work-tape head to the left (not writing anything); change to the state q_{test}.
4. On state q_{test}:
   - If the symbol read from the input-tape is the blank symbol $\square$ and the symbol read from the work-tape is the start symbol $\triangleright$ then write 1 on the output tape and change state to q_{halt}.
   - Otherwise, if the symbols read from the input tape and the work tape are not the same then write 0 on the output tape and change state to q_{halt}.

- Otherwise, if the symbols read from the input tape and the work tape are the same, then move the input-tape head to the right and the work-tape head to the left; stay in the state q_{test}.

As you can see, fully specifying a Turing machine is somewhat tedious and not always very informative. While it is useful to work out one or two examples for yourself (see Exercise 4), in the rest of the book we avoid such overly detailed descriptions and specify TM's in a more high level fashion.

REMARK 1.7
Some texts use as their computational model *single tape* Turing machines, that have one read/write tape that serves as input, work and output tape. This choice does not make any difference for most of this book's results (see Exercise 10). However, Example 1.6 is one exception: it can be shown that such machines require $\Omega(n^2)$ steps to compute the function PAL.

1.2.2 Robustness of our definition.

Most of the specific details of our definition of Turing machines are quite arbitrary. For example, the following simple claims show that restricting the alphabet Γ to be $\{0,1,\square,\triangleright\}$, restricting the machine to have a single work tape, or allowing the tapes to be infinite in both directions will not have a significant effect on the time to compute functions: (Below we provide only proof sketches for these claims; completing these sketches into full proofs is a very good way to gain intuition on Turing machines, see Exercises 5, 6 and 7.)

CLAIM 1.8
For every $f : \{0,1\}^* \to \{0,1\}$ and time-constructible $T : \mathbb{N} \to \mathbb{N}$, if f is computable in time $T(n)$ by a TM M using alphabet Γ then it is computable in time $4\log|\Gamma|T(n)$ by a TM $\tilde{M}$ using the alphabet $\{0,1,\square,\triangleright\}$.

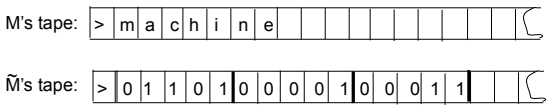

Figure 1.3: We can simulate a machine M using the alphabet $\{\triangleright,\square,\mathtt{a},\mathtt{b},\ldots,\mathtt{z}\}$ by a machine M' using $\{\triangleright,\square,0,1\}$ via encoding every tape cell of M using 5 cells of M'.

PROOF SKETCH: Let M be a TM with alphabet Γ, k tapes, and state set Q that computes the function f in $T(n)$ time. We will show a TM $\tilde{M}$ computing f with alphabet $\{0,1,\square,\triangleright\}$, k tapes and a set Q' of states which will be described below. The idea behind the proof is simple: one can encode any member of Γ using $\log|\Gamma|$ bits.[3] Thus, each of $\tilde{M}$'s work tapes will simply encode one

[3]Recall our conventions that log is taken to base 2, and non-integer numbers are rounded up when necessary.

of M's tapes: for every cell in M's tape we will have $\log|\Gamma|$ cells in the corresponding tape of $\tilde{M}$ (see Figure 1.3).

To simulate one step of M, the machine $\tilde{M}$ will: **(1)** use $\log|\Gamma|$ steps to read from each tape the $\log|\Gamma|$ bits encoding a symbol of Γ **(2)** use its state register to store the symbols read, **(3)** use M's transition function to compute the symbols M writes and M's new state given this information, **(3)** store this information in its state register, and **(4)** use $\log|\Gamma|$ steps to write the encodings of these symbols on its tapes.

One can verify that this can be carried out if $\tilde{M}$ has access to registers that can store M's state, k symbols in Γ and a counter from 1 to k. Thus, there is such a machine $\tilde{M}$ utilizing no more than $10|Q||\Gamma|^k k$ states. (In general, we can always simulate several registers using one register with a larger state space. For example, we can simulate three registers taking values in the sets A,B and C respectively with one register taking a value in the set $A \times B \times C$ which is of size $|A||B||C|$.)

It is not hard to see that for every input $x \in \{0,1\}^n$, if on input x the TM M outputs $f(x)$ within $T(n)$ steps, then $\tilde{M}$ will output the same value within less than $4\log|\Gamma|T(n)$ steps. ■

Claim 1.9

For every $f : \{0,1\}^* \to \{0,1\}$, time-constructible $T : \mathbb{N} \to \mathbb{N}$, if f is computable in time $T(n)$ by a TM M using k tapes (plus additional input and output tapes) then it is computable in time $5kT(n)^2$ by a TM $\tilde{M}$ using only a single work tape (plus additional input and output tapes).

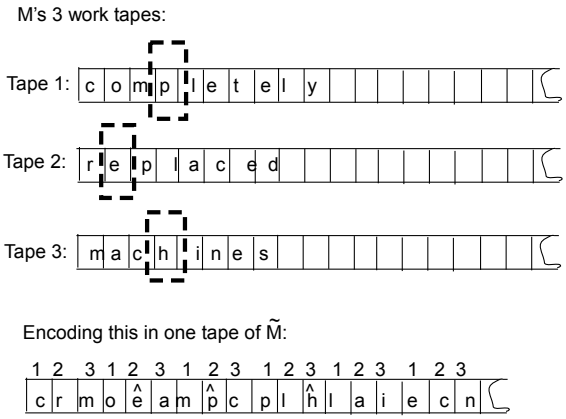

Figure 1.4: Simulating a machine M with 3 work tapes using a machine $\tilde{M}$ with a single work tape (in addition to the input and output tapes).

Proof Sketch: Again the idea is simple: the TM $\tilde{M}$ encodes the k tapes of M on a single tape by using locations $1, k+1, 2k+1, \ldots$ to encode the first tape, locations $2, k+2, 2k+2, \ldots$ to encode the second tape etc.. (see Figure 1.4). For every symbol a in M's alphabet, $\tilde{M}$ will contain both the symbol a and the symbol $\hat{a}$. In the encoding of each tape, exactly one symbol will be of the "$\hat{}$ type", indicating that the corresponding head of M is positioned in that location (see figure). $\tilde{M}$ uses the input and output tape in the same way M does. To simulate one step of M, the machine $\tilde{M}$ makes two sweeps of its work tape: first it sweeps the tape in the left-to-right direction and

records to its register the k symbols that are marked by $\hat{}$. Then $\tilde{M}$ uses M's transition function to determine the new state, symbols, and head movements and sweeps the tape back in the right-to-left direction to update the encoding accordingly. Clearly, $\tilde{M}$ will have the same output as M. Also, since on n-length inputs M never reaches more than location $T(n)$ of any of its tapes, $\tilde{M}$ will never need to reach more than location $kT(n)$ of its work tape, meaning that for each the at most $T(n)$ steps of M, $\tilde{M}$ performs at most $5kT(n)$ work (sweeping back and forth requires about $2T(n)$ steps, and some additional steps may be needed for updating head movement and book keeping). ■

REMARK 1.10
With a bit of care, one can ensure that the proof of Claim 1.9 yields a TM $\tilde{M}$ with the following property: the head movements of $\tilde{M}$ are independent of the contents of its tapes but only on the input length (i.e., $\tilde{M}$ always performs a sequence of left to right and back sweeps of the same form regardless of what is the input). A machine with this property is called *oblivious* and the fact that every TM can be simulated by an oblivious TM will be useful for us later on (see Exercises 8 and 9 and the proof of Theorem 2.10).

CLAIM 1.11
Define a bidirectional *TM to be a TM whose tapes are infinite in* both *directions. For every $f : \{0,1\}^* \to \{0,1\}^*$ and time constructible $T : \mathbb{N} \to \mathbb{N}$, if f is computable in time $T(n)$ by a bidirectional TM M then it is computable in time $4T(n)$ by a standard (unidirectional) TM $\tilde{M}$.*

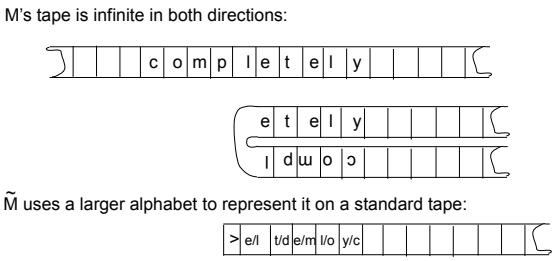

Figure 1.5: To simulate a machine M with alphabet Γ that has tapes infinite in both directions, we use a machine $\tilde{M}$ with alphabet Γ^2 whose tapes encode the "folded" version of M's tapes.

PROOF SKETCH: The idea behind the proof is illustrated in Figure 1.5. If M uses alphabet Γ then $\tilde{M}$ will use the alphabet Γ^2 (i.e., each symbol in $\tilde{M}$'s alphabet corresponds to a pair of symbols in M's alphabet). We encode a tape of M that is infinite in both direction using a standard (infinite in one direction) tape by "folding" it in an arbitrary location, with each location of $\tilde{M}$'s tape encoding two locations of M's tape. At first, $\tilde{M}$ will ignore the second symbol in the cell it reads and act according to M's transition function. However, if this transition function instructs $\tilde{M}$ to go "over the edge" of its tape then instead it will start ignoring the first symbol in each cell and use only the second symbol. When it is in this mode, it will translate left movements into right movements and vice versa. If it needs to go "over the edge" again then it will go back to reading the first symbol of each cell, and translating movements normally. ■

Other changes that will not have a very significant effect include having two or three dimensional tapes, allowing the machine *random access* to its tape, and making the output tape *write only* (see Exercises 11 and 12; also the texts [**?**, **?**] contain more examples). In particular none of these modifications will change the class **P** of polynomial-time computable decision problems defined below in Section 1.5.

1.2.3 The expressive power of Turing machines.

When you encounter Turing machines for the first time, it may not be clear that they do indeed fully encapsulate our intuitive notion of computation. It may be useful to work through some simple examples, such as expressing the standard algorithms for addition and multiplication in terms of Turing machines computing the corresponding functions (see Exercise 4). You can also verify that you can simulate a program in your favorite programming language using a Turing machine. (The reverse direction also holds: most programming languages can simulate a Turing machine.)

EXAMPLE 1.12
(This example assumes some background in computing.) We give a hand-wavy proof that Turing machines can simulate any program written in any of the familiar programming languages such as C or Java. First, recall that programs in these programming languages can be translated (the technical term is *compiled*) into an equivalent *machine language* program. This is a sequence of simple instructions to read from memory into one of a finite number of registers, write a register's contents to memory, perform basic arithmetic operations, such as adding two registers, and control instructions that perform actions conditioned on, say, whether a certain register is equal to zero.

All these operations can be easily simulated by a Turing machine. The memory and register can be implemented using the machine's tapes, while the instructions can be encoded by the machine's transition function. For example, it's not hard to show TM's that add or multiply two numbers, or a two-tape TM that, if its first tape contains a number i in binary representation, can move the head of its second tape to the i^{th} location.

Exercise 13 asks you to give a more rigorous proof of such a simulation for a simple tailor-made programming language.

1.3 Machines as strings and the universal Turing machines.

It is almost obvious that a Turing machine can be represented as a string: since we can write the description of any TM M on paper, we can definitely encode this description as a sequence of zeros and ones. Yet this simple observation— that we can treat programs as data— has had far reaching consequences on both the theory and practice of computing. Without it, we would not have had *general purpose* electronic computers, that, rather than fixed to performing one task, can execute arbitrary programs.

Because we will use this notion of representing TM's as strings quite extensively, it may be worth to spell out our representation out a bit more concretely. Since the behavior of a Turing machine is determined by its transition function, we will use the list of all inputs and outputs of this function (which can be easily encoded as a string in $\{0,1\}^*$) as the encoding of the Turing machine.[4] We will also find it convenient to assume that our representation scheme satisfies the following properties:

1. Every string in $\{0,1\}^*$ represents *some* Turing machine.

   This is easy to ensure by mapping strings that are not valid encodings into some canonical trivial TM, such as the TM that immediately halts and outputs zero on any input.

2. Every TM is represented by infinitely many strings.

   This can be ensured by specifying that the representation can end with an arbitrary number of 1's, that are ignored. This has somewhat similar effect as the *comments* of many programming languages (e.g., the `/*...*/` construct in C,C++ and Java) that allows to add superfluous symbols to any program.

If M is a Turing machine, then we use $\llcorner M \lrcorner$ to denotes M's representation as a binary string. If α is a string then we denote the TM that α represents by M_α. As is our convention, we will also often use M to denote both the TM and its representation as a string. Exercise 14 asks you to fully specify a representation scheme for Turing machines with the above properties.

1.3.1 The Universal Turing Machine

It was Turing that first observed that general purpose computers are possible, by showing a *universal* Turing machine that can *simulate* the execution of every other TM M given M's description as input. Of course, since we are so used to having a universal computer on our desktops or even in our pockets, today we take this notion for granted. But it is good to remember why it was once counterintuitive. The parameters of the universal TM are fixed —alphabet size, number of states, and number of tapes. The corresponding parameters for the machine being simulated could be much larger. The reason this is not a hurdle is, of course, the ability to use *encodings*. Even if the universal TM has a very simple alphabet, say $\{0,1\}$, this is sufficient to allow it to represent the other machine's state and and transition table on its tapes, and then follow along in the computation step by step.

Now we state a computationally efficient version of Turing's construction due to Hennie and Stearns [**?**]. To give the essential idea we first prove a slightly relaxed variant where the term $T \log T$ below is replaced with T^2. But since the efficient version is needed a few times in the book, a full proof is also given at the end of the chapter (see Section 1.A).

[4] Note that the size of the alphabet, the number of tapes, and the size of the state space can be deduced from the transition function's table. We can also reorder the table to ensure that the special states $q_{\mathsf{start}}, q_{\mathsf{halt}}$ are the first 2 states of the TM. Similarly, we may assume that the symbols $\triangleright, \square, 0, 1$ are the first 4 symbols of the machine's alphabet.

THEOREM 1.13 (EFFICIENT UNIVERSAL TURING MACHINE)
There exists a TM $\mathcal{U}$ such that for every $x, \alpha \in \{0,1\}^*$, $\mathcal{U}(x,\alpha) = M_\alpha(x)$, where M_α denotes the TM represented by α.
Furthermore, if M_α halts on input x within T steps then $\mathcal{U}(x,\alpha)$ halts within $CT \log T$ steps, where C is a number independent of $|x|$ and depending only on M_α's alphabet size, number of tapes, and number of states.

REMARK 1.14
A common exercise in programming courses is to write an *interpreter* for a particular programming language using the same language. (An interpreter takes a program P as input and outputs the result of executing the program P.) Theorem 1.13 can be considered a variant of this exercise.

PROOF: Our universal TM $\mathcal{U}$ is given an input x, α, where α represents some TM M, and needs to output $M(x)$. A crucial observation is that we may assume that M **(1)** has a single work tape (in addition to the input and output tape) and **(2)** uses the alphabet $\{\triangleright, \square, 0, 1\}$. The reason is that $\mathcal{U}$ can transform a representation of every TM M into a representation of an equivalent TM $\tilde{M}$ that satisfies these properties as shown in the proofs of Claims 1.8 and 1.9. Note that these transformations may introduce a quadratic slowdown (i.e., transform M from running in T time to running in $C'T^2$ time where C' depends on M's alphabet size and number of tapes).

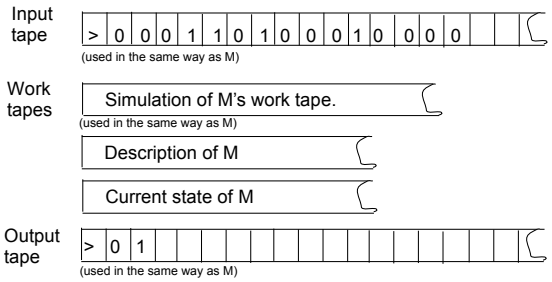

Figure 1.6: The universal TM $\mathcal{U}$ has in addition to the input and output tape, three work tapes. One work tape will have the same contents as the simulated machine M, another tape includes the description M (converted to an equivalent one-work-tape form), and another tape contains the current state of M.

The TM $\mathcal{U}$ uses the alphabet $\{\triangleright, \square, 0, 1\}$ and three work tapes in addition to its input and output tape (see Figure 1.6). $\mathcal{U}$ uses its input tape, output tape, and one of the work tapes in the same way M uses its three tapes. In addition, $\mathcal{U}$ will use its first extra work tape to store the table of values of M's transition function (after applying the transformations of Claims 1.8 and 1.9 as noted above), and its other extra work tape to store the current state of M. To simulate one computational step of M, $\mathcal{U}$ scans the table of M's transition function and the current state to find out the new state, symbols to be written and head movements, which it then executes. We see that each computational step of M is simulated using C steps of $\mathcal{U}$, where C is some number depending on the size of the transition function's table.

This high level description can turned into an exact specification of the TM $\mathcal{U}$, though we leave this to the reader. If you are not sure how this can be done, think first of how you would program these steps in your favorite programming language and then try to transform this into a description of a Turing machine. ■

REMARK 1.15
It is sometimes useful to consider a variant of the universal TM $\mathcal{U}$ that gets a number t as an extra input (in addition to x and α), and outputs $M_\alpha(x)$ if and only if M_α halts on x within t steps (otherwise outputting some special failure symbol). By adding a counter to $\mathcal{U}$, the proof of Theorem 1.13 can be easily modified to give such a universal TM with the same efficiency.

1.4 Uncomputable functions.

It may seem "obvious" that every function can be computed, given sufficient time. However, this turns out to be false: there exist functions that cannot be computed within any finite number of steps!

THEOREM 1.16
There exists a function $\mathsf{UC} : \{0,1\}^* \to \{0,1\}$ *that is not computable by any TM.*

PROOF: The function UC is defined as follows: for every $\alpha \in \{0,1\}^*$, let M be the TM represented by α. If on input α, M halts within a finite number of steps and outputs 1 then $\mathsf{UC}(\alpha)$ is equal to 0, otherwise $\mathsf{UC}(\alpha)$ is equal to 1.

Suppose for the sake of contradiction that there exists a TM M such that $M(\alpha) = \mathsf{UC}(\alpha)$ for every $\alpha \in \{0,1\}^*$. Then, in particular, $M(\llcorner M \lrcorner) = \mathsf{UC}(\llcorner M \lrcorner)$. But this is impossible: by the definition of UC, if $\mathsf{UC}(\llcorner M \lrcorner) = 1$ then $M(\llcorner M \lrcorner)$ cannot be equal to 1, and if $\mathsf{UC}(\llcorner M \lrcorner) = 0$ then $M(\llcorner M \lrcorner)$ cannot be equal to 0. This proof technique is called "diagnoalization", see Figure 1.7. ■

1.4.1 The Halting Problem

One might ask why should we care whether or not the function UC described above is computable—why would anyone want to compute such a contrived function anyway? We now show a more natural uncomputable function. The function HALT takes as input a pair α, x and outputs 1 if and only if the TM M_α represented by α halts on input x within a finite number of steps. This is definitely a function we want to compute: given a computer program and an input we'd certainly like to know if the program is going to enter an infinite loop on this input. Unfortunately, this is not possible, even if we were willing to wait an arbitrary long time:

THEOREM 1.17
HALT *is not computable by any TM.*

PROOF: Suppose, for the sake of contradiction, that there was a TM M_{HALT} computing HALT. We will use M_{HALT} to show a TM M_{UC} computing UC, contradicting Theorem 1.16.

The TM M_{UC} is simple: on input α, we run $M_{\mathsf{HALT}}(\alpha, \alpha)$. If the result is 0 (meaning that the machine represented by α does not halt on α) then we output 1. Otherwise, we use the universal

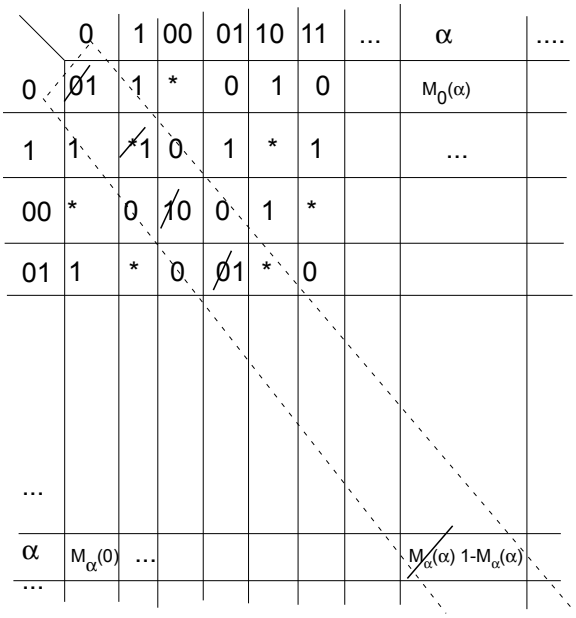

Figure 1.7: Suppose we order all strings in lexicographic order, and write in a table the value of $M_\alpha(x)$ for all strings α, x, where M_α denotes the TM represented by the string α and we use $\star$ to denote the case that $M_\alpha(x)$ is not a value in $\{0,1\}$ or that M_α does not halt on input x. Then, UC is defined by "negating" the diagonal of this table, and by its definition it cannot be computed by any TM.

TM $\mathcal{U}$ to compute $M(\alpha)$, where M is the TM represented by α. If $M(\alpha) = 0$ we output 1, and otherwise we output 1. Note that indeed, under the assumption that $M_{\mathsf{HALT}}(\alpha, x)$ outputs within a finite number of steps $\mathsf{HALT}(\alpha, x)$, the TM $M_{\mathsf{UC}}(\alpha)$ will output $\mathsf{UC}(\alpha)$ within a finite number of steps. ■

REMARK 1.18
The proof technique employed to show Theorem 1.17— namely showing that HALT is uncomputable by showing an algorithm for UC using a hypothetical algorithm for HALT— is called a *reduction*. We will see many reductions in this book, often used (as is the case here) to show that a problem B is at least as hard as a problem A, by showing an algorithm that could solve A given a procedure that solves B.

There are many other examples for interesting uncomputable (also known as *undecidable*) functions, see Exercise 15. There are even uncomputable functions whose formulation has seemingly nothing to do with Turing machines or algorithms. For example, the following problem cannot be solved in finite time by any TM: given a set of polynomial equations with integer coefficients, find out whether these equations have an integer solution (i.e., whether there is an assignment of integers to the variables that satisfies the equations). This is known as the problem of solving Diophantine equations, and in 1900 Hilbert mentioned finding such algorithm to solve it (which he presumed to exist) as one of the top 23 open problems in mathematics.

For more on computability theory, see the chapter notes and the book's website.

1.5 Deterministic time and the class P.

A *complexity class* is a set of functions that can be computed within a given resource. We will now introduce our first complexity classes. For reasons of technical convenience, throughout most of this book we will pay special attention to Boolean functions (that have one bit output), also known as *decision problems* or *languages*. (Recall that we identify a Boolean function f with the language $L_f = \{x : f(x) = 1\}$.)

DEFINITION 1.19 (THE CLASS **DTIME**.)
Let $T : \mathbb{N} \to \mathbb{N}$ be some function. We let $\mathbf{DTIME}(T(n))$ be the set of all Boolean (one bit output) functions that are computable in $c \cdot T(n)$-time for some constant $c > 0$.

The following class will serve as our rough approximation for the class of decision problems that are efficiently solvable.

DEFINITION 1.20 (THE CLASS **P**)
$\mathbf{P} = \cup_{c \geq 1} \mathbf{DTIME}(n^c)$

Thus, we can phrase the question from the introduction as to whether the dinner party problem has an efficient algorithm as follows: *"Is* INDSET *in* **P***?"*, where INDSET is the language defined in Example 1.6.

1.5.1 On the philosophical importance of P

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

Note that the class **P** is useful only in a certain context. Turing machines are a poor model if one is designing algorithms that must run in a fraction of a second on the latest PC (in which case one must carefully account for fine details about the hardware). However, if the question is whether any subexponential algorithms exist for say INDSET then even an n^{20} algorithm would be a fantastic breakthrough.

P is also a natural class from the viewpoint of a programmer. Suppose undergraduate programmers are asked to invent the definition of an "efficient" computation. Presumably, they would agree that a computation that runs in linear or quadratic time is "efficient." Next, since programmers often write programs that call other programs (or subroutines), they might find it natural to consider a program "efficient" if it performs only "efficient" computations and calls subroutines that are "efficient". The notion of "efficiency" obtained turns out to be exactly the class **P** [**?**].

1.5.2 Criticisms of P and some efforts to address them

Now we address some possible criticisms of the definition of **P**, and some related complexity classes that address these.

Worst-case exact computation is too strict. The definition of **P** only considers algorithms that compute the function *exactly* on *every* possible input. However, not all possible inputs arise in practice (although it's not always easy to characterize the inputs that do). Chapter 15 gives a theoretical treatment of *average-case complexity* and defines the analogue of **P** in that context. Sometimes, users are willing to settle for *approximate* solutions. Chapter 18 contains a rigorous treatment of the complexity of approximation.

Other physically realizable models. If we were to make contact with an advanced alien civilization, would their class **P** be any different from the class defined here?

Most scientists believe the *Church-Turing (CT) thesis*, which states that every physically realizable computation device— whether it's silicon-based, DNA-based, neuron-based or using some alien technology— can be simulated by a Turing machine. Thus they believe that the set of *computable* problems would be the same for aliens as it is for us. (The CT thesis is not a theorem, merely a belief about the nature of the world.)

However, when it comes to *efficiently* computable problems, the situation is less clear. The **strong form of the CT thesis** says that every physically realizable computation model can be simulated by a TM *with polynomial overhead* (in other words, t steps on the model can be simulated in t^c steps on the TM, where c is a constant that depends upon the model). If true, it implies that the class **P** defined by the aliens will be the same as ours. However, several objections have been made to this strong form.

(a) *Issue of precision:* TM's compute with discrete symbols, whereas physical quantities may be real numbers in $\mathbb{R}$. Thus TM computations may only be able to approximately simulate the real world. Though this issue is not perfectly settled, it seems so far that TMs do not suffer from an inherent handicap. After all, real-life devices suffer from noise, and physical quantities can only be measured up to finite precision. Thus a TM could simulate the real-life device using finite precision. (Note also that we often only care about the *most significant bit* of the result, namely, a 0/1 answer.)

Even so, in Chapter 14 we also consider a modification of the TM model that allows computations in $\mathbb{R}$ as a basic operation. The resulting complexity classes have fascinating connections with the usual complexity classes.

(b) *Use of randomness:* The TM as defined is *deterministic*. If randomness exists in the world, one can conceive of computational models that use a source of random bits (i.e., "coin tosses"). Chapter 7 considers Turing Machines that are allowed to also toss coins, and studies the class **BPP**, that is the analogue of **P** for those machines. (However, we will see in Chapters 16 and 17 the intriguing possibility that randomized computation may be no more powerful than deterministic computation.)

(c) *Use of quantum mechanics:* A more clever computational model might use some of the counterintuitive features of quantum mechanics. In Chapter 20 we define the class **BQP**,

that generalizes **P** in such a way. We will see problems in **BQP** that are currently not known to be in **P**. However, currently it is unclear whether the quantum model is truly physically realizable. Even if it is realizable it currently seems only able to efficiently solve only very few "well-structured" problems that are (presumed to be) not in **P**. Hence insights gained from studying **P** could still be applied to **BQP**.

(d) *Use of other exotic physics, such as string theory.* Though an intriguing possibility, it hasn't yet had the same scrutiny as quantum mechanics.

Decision problems are too limited. Some computational problems are not easily expressed as decision problems. Indeed, we will introduce several classes in the book to capture tasks such as computing non-Boolean functions, solving search problems, approximating optimization problems, interaction, and more. Yet the framework of decision problems turn out to be surprisingly expressive, and we will often use it in this book.

1.5.3 Edmonds' quote

We conclude this section with a quote from Edmonds [**?**], that in the paper showing a polynomial-time algorithm for the maximum matching problem, explained the meaning of such a result as follows:

> *For practical purposes computational details are vital. However, my purpose is only to show as attractively as I can that there is an efficient algorithm. According to the dictionary, "efficient" means "adequate in operation or performance." This is roughly the meaning I want in the sense that it is conceivable for maximum matching to have no efficient algorithm.*
>
> *...There is an obvious finite algorithm, but that algorithm increases in difficulty exponentially with the size of the graph. It is by no means obvious whether or not there exists an algorithm whose difficulty increases only algebraically with the size of the graph.*
>
> *...When the measure of problem-size is reasonable and when the sizes assume values arbitrarily large, an asymptotic estimate of ... the order of difficulty of an algorithm is theoretically important. It cannot be rigged by making the algorithm artificially difficult for smaller sizes.*
>
> *...One can find many classes of problems, besides maximum matching and its generalizations, which have algorithms of exponential order but seemingly none better ... For practical purposes the difference between algebraic and exponential order is often more crucial than the difference between finite and non-finite.*
>
> *...It would be unfortunate for any rigid criterion to inhibit the practical development of algorithms which are either not known or known not to conform nicely to the criterion. Many of the best algorithmic idea known today would suffer by such theoretical pedantry. ... However, if only to motivate the search for good, practical algorithms, it is important to realize that it is mathematically sensible even to question their existence. For one thing the task can then be described in terms of concrete conjectures.*

WHAT HAVE WE LEARNED?

- There are many equivalent ways to mathematically model computational processes; we use the standard Turing machine formalization.
- Turing machines can be represented as strings. There is a *universal* TM that can emulate (with small overhead) any TM given its representation.
- There exist functions, such as the Halting problem, that cannot be computed by any TM regardless of its running time.
- The class **P** consists of all decision problems that are solvable by Turing machines in polynomial time. We say that problems in **P** are efficiently solvable.
- All low-level choices (number of tapes, alphabet size, etc..) in the definition of Turing machines are immaterial, as they will not change the definition of **P**.

Chapter notes and history

Although certain algorithms have been studied for thousands of years, and some forms of computing devices were designed before the 20th century (most most notably Charles Babbage's difference and analytical engines in the mid 1800's), it seems fair to say that the foundations of modern computer science were only laid in the 1930's.

In 1931, Kurt Gödel shocked the mathematical world by showing that certain true statements about the natural numbers are *inherently unprovable*, thereby shattering an ambitious agenda set in 1900 by David Hilbert to base all of mathematics on solid axiomatic foundations. In 1936, Alonzo Church defined a model of computation called λ-calculus (which years later inspired the programming language LISP) and showed the existence of functions *inherently uncomputable* in this model [**?**]. A few months later, Alan Turing independently introduced his Turing machines and showed functions inherently uncomputable by such machines [**?**]. Turing also introduced the idea of the *universal* Turing machine that can be loaded with arbitrary programs. The two models turned out to be equivalent, but in the words of Church himself, Turing machines have "the advantage of making the identification with effectiveness in the ordinary (not explicitly defined) sense evident immediately". The anthology [**?**] contains many of the seminal papers in the theory of computability. Part II of Sipser's book [**?**] is a good gentle introduction to this theory, while the books [**?**, **?**, **?**] go into a bit more depth. This book's web site also contains some additional coverage of this theory.

During World War II Turing designed mechanical code-breaking devices and played a key role in the effort to crack the German "Enigma" cipher, an achievement that had a decisive effect on the war's progress (see the biographies [**?**, **?**]).[5] After World War II, efforts to build electronic universal computers were undertaken in both sides of the Atlantic. A key figure in these efforts was John

[5] Unfortunately, Turing's wartime achievements were kept confidential during his lifetime, and so did not keep him from being forced by British courts to take hormones to "cure" his homosexuality, resulting in his suicide in 1954.

von-Neumann, an extremely prolific scientist that was involved in anything from the Manhattan project to founding game theory in economics. To this day essentially all digital computers follow the "von-Neumann architecture" he pioneered while working on the design of the EDVAC, one of the earliest digital computers [**?**].

As computers became more prevalent, the issue of efficiency in computation began to take center stage. Cobham [**?**] defined the class **P** and suggested it may be a good formalization for efficient computation. A similar suggestion was made by Edmonds ([**?**], see quote above) in the context of presenting a highly non-trivial polynomial-time algorithm for finding a maximum matching in general graphs. Hartmanis and Stearns [**?**] defined the class $\mathbf{DTIME}(T(n))$ for every function T, and proved the slightly relaxed version of Theorem 1.13 we showed above (the version we stated and prove below was given by Hennie and Stearns [**?**]). They also coined the name "computational complexity" and proved an interesting "speed-up theorem": if a function f is computable by a TM M in time $T(n)$ then for every constant $c \geq 1$, f is computable by a TM $\tilde{M}$ (possibly with larger state size and alphabet size than M) in time $T(n)/c$. This speed-up theorem is another justification for ignoring constant factors in the definition of $\mathbf{DTIME}(T(n))$. Blum [**?**] suggested an axiomatic formalization of complexity theory, that does not explicitly mention Turing machines.

We have omitted a discussion of some of the "bizarre conditions" that may occur when considering time bounds that are not time-constructible, especially "huge" time bounds (i.e., function $T(n)$ that are much larger than exponential in n). For example, there is a non-time constructible function $T : \mathbb{N} \to \mathbb{N}$ such that every function computable in time $T(n)$ can also be computed in the much shorter time $\log T(n)$. However, we will not encounter non time-constructible time bounds in this book.

Exercises

§1 For each of the following pairs of functions f, g determine whether $f = o(g)$, $g = o(f)$ or $f = \Theta(g)$. If $f = o(g)$ then find the first number n such that $f(n) < g(n)$:

(a) $f(n) = n^2$, $g(n) = 2n^2 + 100\sqrt{n}$.
(b) $f(n) = n^{100}$, $g(n) = 2^{n/100}$.
(c) $f(n) = n^{100}$, $g(n) = 2^{n^{1/100}}$.
(d) $f(n) = \sqrt{n}, g(n) = 2^{\sqrt{\log n}}$.
(e) $f(n) = n^{100}, g(n) = 2^{(\log n)^2}$.
(f) $f(n) = 1000n, g(n) = n \log n$.

§2 For each of the following recursively defined functions f, find a closed (non-recursive) expression for a function g such that $f(n) = \Theta(g(n))$.

(Note: below we only supply the recursive rule, you can assume that $f(1) = f(2) = \cdots = f(10) = 1$ and the recursive rule is applied for $n > 10$; in any case regardless how the base case it won't make any difference to the answer - can you see why?)

(a) $f(n) = f(n-1) + 10$.

(b) $f(n) = f(n-1) + n$.

(c) $f(n) = 2f(n-1)$.

(d) $f(n) = f(n/2) + 10$.

(e) $f(n) = f(n/2) + n$.

(f) $f(n) = 2f(n/2) + n$.

(g) $f(n) = 3f(n/2)$.

§3 The MIT museum contains a kinetic sculpture by Arthur Ganson called "Machine with concrete" (see Figure 1.8). It consists of 13 gears connected to one another in a series such that each gear moves 50 times slower than the previous one. The fastest gear is constantly rotated by an engine at a rate of 212 rotations per minute. The slowest gear is fixed to a block of concrete and so apparently cannot move at all. How come this machine does not break apart?

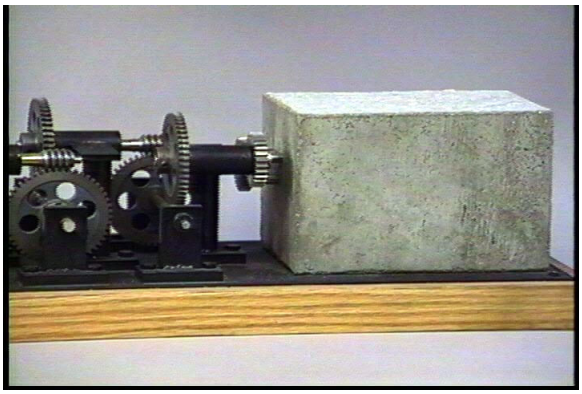

Figure 1.8: Machine with concrete by Arthur Ganson.

§4 Let f be the *addition* function that maps the representation of a pair of numbers x, y to the representation of the number $x + y$. Let g be the *multiplication* function that maps $\langle x, y \rangle$ to $\llcorner x \cdot y \lrcorner$. Prove that both f and g are computable by writing down a full description (including the states, alphabet and transition function) of the corresponding Turing machines.

Hint: Follow the grade-school algorithms.

§5 Complete the proof of Claim 1.8 by writing down explicitly the description of the machine $\tilde{M}$.

§6 Complete the proof of Claim 1.9.

§7 Complete the proof of Claim 1.11.

§8 Define a TM M to be *oblivious* if its head movement does not depend on the input but only on the input length. That is, M is oblivious if for every input $x \in \{0,1\}^*$ and $i \in \mathbb{N}$, the location of each of M's heads at the i^{th} step of execution on input x is only a function of $|x|$

and i. Show that for every time-constructible $T : \mathbb{N} \to \mathbb{N}$, if $L \in \mathbf{DTIME}(T(n))$ then there is an oblivious TM that decides L in time $O(T(n)^2)$. Furthermore, show that there is such a TM that uses only *two tapes*: one input tape and one work/output tape.

Hint: Use the proof of Claim 1.9.

§9 Show that for every time-constructible $T : \mathbb{N} \to \mathbb{N}$, if $L \in \mathbf{DTIME}(T(n))$ then there is an oblivious TM that decides L in time $O(T(n) \log T(n))$.

Hint: show that the universal TM $\mathcal{U}$ obtained by the proof of Theorem 1.13 can be tweaked to be oblivious.

§10 Define a *single-tape* Turing machine to be a TM that has only one read/write tape, that is used as input, work and output tape. Show that for every (time-constructible) $T : \mathbb{N} \to \mathbb{N}$ and $f \in \mathbf{DTIME}(T(n))$, f can be computed in $O(T(n)^2)$ steps by a single-tape TM.

§11 Define a *two dimensional* Turing machine to be a TM where each of its tapes is an infinite grid (and the machine can move not only Left and Right but also Up and Down). Show that for every (time-constructible) $T : \mathbb{N} \to \mathbb{N}$ and every Boolean function f, if g can be computed in time $T(n)$ using a two-dimensional TM then $f \in \mathbf{DTIME}(T(n)^2)$.

§12 Define a *RAM Turing machine* to be a Turing machine that has *random access memory*. We formalize this as follows: the machine has additional two symbol on its alphabet we denote by `R` and `W` and an additional state we denote by q_{access}. We also assume that the machine has an infinite array A that is initialized to all blanks. Whenever the machine enters q_{access}, if its address tape contains $\llcorner i \lrcorner$`R` (where $\llcorner i \lrcorner$ denotes the binary representation of i) then the value $A[i]$ is written in the cell next to the `R` symbol. If its tape contains $\llcorner i \lrcorner$`W`σ (where σ is some symbol in the machine's alphabet) then $A[i]$ is set to the value σ.

Show that if a Boolean function f is computable within time $T(n)$ (for some time-constructible T) by a RAM TM, then it is in $\mathbf{DTIME}(T(n)^2)$.

§13 Consider the following simple programming language. It has a single infinite array `A` of elements in $\{0, 1, \square\}$ (initialized to $\square$) and a single integer variable `i`. A program in this language contains a sequence of lines of the following form:

label : `If A[i] equals` σ `then` *cmds*

Where $\sigma \in \{0, 1, \square\}$ and *cmds* is a list of one or more of the following commands: **(1)** `Set A[i] to` τ where $\tau \in \{0, 1, \square\}$, **(2)** `Goto` *label*, **(3)** `Increment i by one`, **(4)** `Decrement i by one`, and **(5)** `Output` b `and halt.` where $b \in \{0, 1\}$. A program is executed on an input $x \in \{0, 1\}^n$ by placing the i^{th} bit of x in `A[i]` and then running the program following the obvious semantics.

Prove that for every functions $f : \{0, 1\}^* \to \{0, 1\}$ and (time constructible) $T : \mathbb{N} \to \mathbb{N}$, if f is computable in time $T(n)$ by a program in this language, then $f \in \mathbf{DTIME}(T(n))$.

§14 Give a full specification of a representation scheme of Turing machines as binary string strings. That is, show a procedure that transforms any TM M (e.g., the TM computing the function PAL described in Example 1.6) into a binary string $\llcorner M \lrcorner$. It should be possible to recover M from $\llcorner M \lrcorner$, or at least recover a functionally equivalent TM (i.e., a TM $\tilde{M}$ computing the same function as M with the same running time).

§15 A *partial* function from $\{0,1\}^*$ to $\{0,1\}^*$ is a function that is not necessarily defined on all its inputs. We say that a TM M computes a partial function f if for every x on which f is defined, $M(x) = f(x)$ and for every x on which f is not defined M gets into an infinite loop when executed on input x. If $\mathcal{S}$ is a set of partial functions, we define $f_{\mathcal{S}}$ to be the Boolean function that on input α outputs 1 iff M_α computes a partial function in $\mathcal{S}$. *Rice's Theorem* says that for every non-trivial $\mathcal{S}$ (a set that is not the empty set nor the set of all partial functions), the $f_{\mathcal{S}}$ is not computable.

(a) Show that Rice's Theorem yields an alternative proof for Theorem 1.17 by showing that the function HALT is not computable.

(b) Prove Rice's Theorem.

Hint: By possible changing from $\mathcal{S}$ to its complement, we may assume that the empty function $\emptyset$ (that is not defined on any input) is in $\mathcal{S}$ there is some function f that is defined on some input x that is not in $\mathcal{S}$. Use this to show that an algorithm to compute $f_{\mathcal{S}}$ can compute the function HALT_x which outputs 1 on input α iff M_α halts on input x. Then reduce computing HALT to computing HALT_x thereby deriving Rice's Theorem from Theorem 1.17.

§16 Prove that the following languages/decision problems on graphs are in **P**: (You may pick either the adjacency matrix or adjacency list representation for graphs; it will not make a difference. Can you see why?)

(a) CONNECTED — the set of all connected graphs. That is, $G \in$ CONNECTED if every pair of vertices u, v in G are connected by a path.

(b) TRIANGLEFREE — the set of all graphs that do not contain a triangle (i.e., a triplet u, v, w of connected distinct vertices.

(c) BIPARTITE — the set of all bipartite graphs. That is, $G \in$ BIPARTITE if the vertices of G can be partitioned to two sets A, B such that all edges in G are from a vertex in A to a vertex in B (there is no edge between two members of A or two members of B).

(d) TREE — the set of all trees. A graph is a *tree* if it is connected and contains no cycles. Equivalently, a graph G is a tree if every two distinct vertices u, v in G are connected by exactly one simple path (a path is simple if it has no repeated vertices).

§17 Recall that normally we assume that numbers are represented as string using the *binary* basis. That is, a number n is represented by the sequence $x_0, x_1, \ldots, x_{\log n}$ such that $n = \sum_{i=0}^{n} x_i 2^i$. However, we could have used other encoding schemes. If $n \in \mathbb{N}$ and $b \geq 2$, then

the representation of n in base b, denoted by $\llcorner n \lrcorner_b$ is obtained as follows: first represent n as a sequence of digits in $\{0, \ldots, b-1\}$, and then replace each digit by a sequence of zeroes and ones. The *unary* representation of n, denoted by $\llcorner n \lrcorner_{\text{unary}}$ is the string 1^n (i.e., a sequence of n ones).

(a) Show that choosing a different base of representation will make no difference to the class **P**. That is, show that for every subset S of the natural numbers, if we define $L^b_S = \{ \llcorner n \lrcorner_b : n \in S\}$ then for every $b \geq 2$, $L^b_S \in \mathbf{P}$ iff $L^2_S \in \mathbf{P}$.

(b) Show that choosing the unary representation make make a difference by showing that the following language is in **P**:

$$\mathsf{UNARYFACTORING} = \{\langle \llcorner n \lrcorner_{\text{unary}}, \llcorner \ell \lrcorner_{\text{unary}}, \llcorner k \lrcorner_{\text{unary}} \rangle : \text{there is } j \in (\ell, k) \text{ dividing } n\}$$

It is not known to be in **P** if we choose the binary representation (see Chapters 10 and 20). In Chapter 3 we will see that there is a problem that is *proven* to be in **P** when choosing the unary representation but not in **P** when using the binary representation.

1.A Proof of Theorem 1.13: Universal Simulation in $O(T \log T)$-time

We now show how to prove Theorem 1.13 as stated. That is, we show a universal TM $\mathcal{U}$ such that given an input x and a description of a TM M that halts on x within T steps, $\mathcal{U}$ outputs $M(x)$ within $O(T \log T)$ time (where the constants hidden in the O notation may depend on the parameters of the TM M being simulated).

The general structure of $\mathcal{U}$ will be as in Section 1.3.1, using the input and output tape as M does, and with extra work tapes to store M's transition table and current state. We will also have another "scratch" work tape to assist in certain computation. The main obstacle we need to overcome is that we cannot use Claim 1.9 to reduce the number of M's work tapes to one, as that claim introduces too much of overhead in the simulation. Therefore, we need to show a different way to encode all of M's work tapes using one tape of $\mathcal{U}$.

Let k be the number of tapes that M uses and Γ its alphabet. Following the proof of Claim 1.8, we may assume that $\mathcal{U}$ uses the alphabet Γ^k (as this can be simulated with a overhead depending only on $|\Gamma|$). Thus we can encode in each cell of $\mathcal{U}$'s work tape k symbols of Γ, each corresponding to a symbol from one of M's tapes. However, we still have to deal with the fact that M has k read/write heads that can each move independently to the left or right, whereas $\mathcal{U}$'s work tape only has a single head. Paraphrasing the famous saying, our strategy to handle this can be summarized as follows:

"If Muhammad will not come to the mountain then the mountain will go to Muhammad".

That is, since we can not move $\mathcal{U}$'s read/write head in different directions at once, we simply move the tape "under" the head. To be more specific, since we consider $\mathcal{U}$'s alphabet to be Γ^k, we can think of $\mathcal{U}$'s main work tape not as a single tape but rather k *parallel tapes*; that is, we can

think of $\mathcal{U}$ as having k tapes with the property that in each step either all their read/write heads go in unison one location to the left or they all go one location to the right (see Figure 1.9).

To simulate a single step of M we shift all the non-blank symbols in each of these parallel tapes until the head's position in these parallel tapes corresponds to the heads' positions of M's k tapes. For example, if $k = 3$ and in some particular step M's transition function specifies the movements L, R, R then $\mathcal{U}$ will shift all the non-blank entries of its first parallel tape one cell to the right, and shift the non-blank entries of its second and third tapes one cell to the left. ($\mathcal{U}$ can easily perform these shifts using the additional "scratch" work tape.)

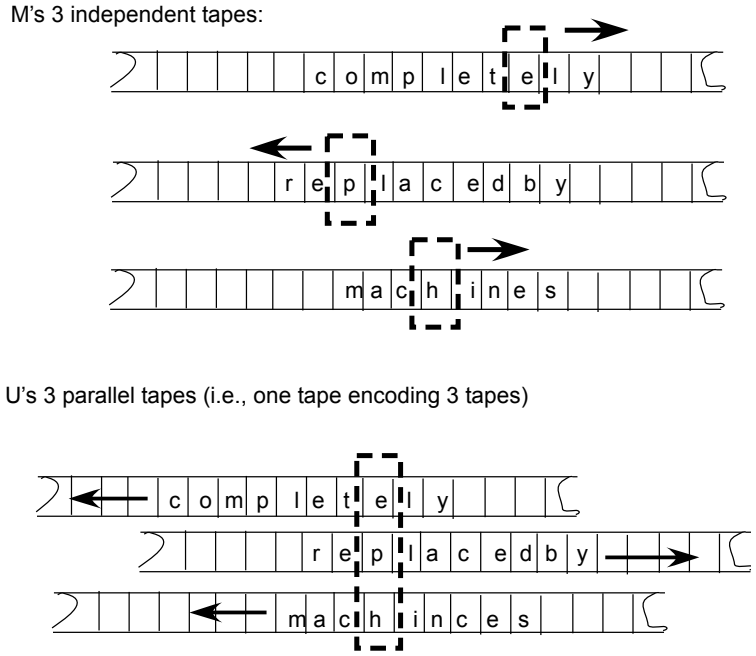

Figure 1.9: Packing k tapes of M into one tape of $\mathcal{U}$. We consider $\mathcal{U}$'s single work tape to be composed of k parallel tapes, whose heads move in unison, and hence we shift the contents of these tapes to simulate independent head movement.

The approach above is still not good enough to get $O(T \log T)$-time simulation. The reason is that there may be as much as T non-blank symbols in each tape, and so each shift operation may cost $\mathcal{U}$ at least T operations per each step of M. Our approach to deal with this is to create "buffer zones": rather than having each of $\mathcal{U}$'s parallel tapes correspond exactly to a tape of M, we add a special kind of blank symbol $\boxtimes$ to the alphabet of $\mathcal{U}$'s parallel tapes with the semantics that this symbol is ignored in the simulation. For example, if the non-blank contents of M's tape are 010 then this can be encoded in the corresponding parallel tape of $\mathcal{U}$ not just by 010 but also by $0\boxtimes01$ or $0\boxtimes\boxtimes1\boxtimes0$ and so on.

For convenience, we think of $\mathcal{U}$'s parallel tapes as infinite in both the left and right directions (this can be easily simulated with minimal overhead, see Claim 1.11). Thus, we index their locations by $0, \pm1, \pm2, \ldots$. Normally we keep $\mathcal{U}$'s head on location 0 of these parallel tapes. We will only

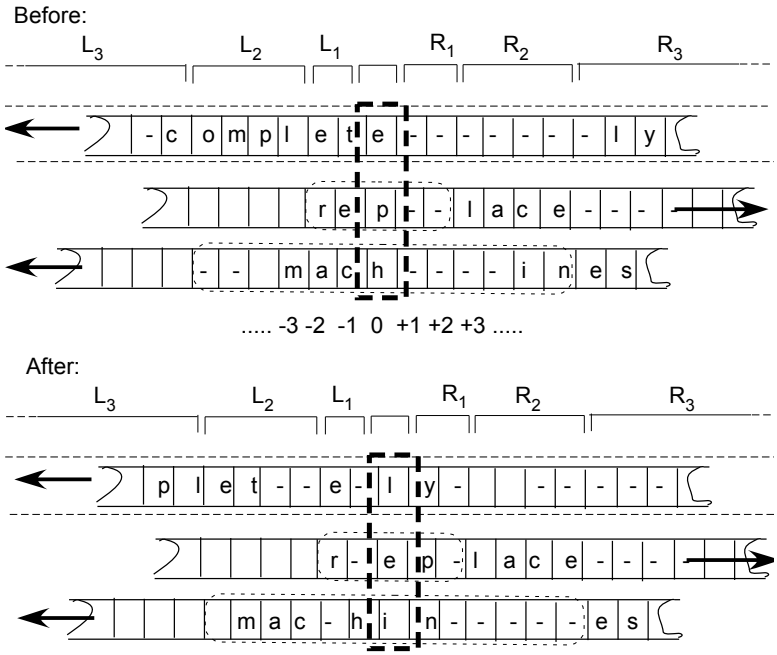

Figure 1.10: Performing a shift of the parallel tapes. The left shift of the first tape involves zones $L_1, R_1, L_2, R_2, L_3, R_3$, the right shift of the second tape involves only L_1, R_1, while the left shift of the third tape involves zones L_1, R_1, L_2, R_2. We maintain the invariant that each zone is either empty, half-full or full. Note that - denotes ⊡.

move it temporarily to perform a shift when, following our general approach, we simulate a left head movement by shifting the tape to the right and vice versa. At the end of the shift we return the head to location 0.

We split the tapes into zones $L_0, R_0, L_1, R_1, \ldots$ (we'll only need to go up to $L_{\log T+1}, R_{\log T+1}$) where zone L_i contains the 2^i cells in the interval $[2^i..2^{i+1}-1]$ and zone R_i contains the cells in the interval $[-2^{i+1}+1..-2^i]$ (location 0 is not in any zone). We shall always maintain the following invariants:

- Each of the zones is either *empty*, *full*, or *half-full* with non-⊡ symbols. That is, the number of symbols in zone L_i that are not ⊡ is either 0,2^{i-1}, or 2^i and the same holds for R_i. (We treat the ordinary □ symbol the same as any other symbol in Γ and in particular a zone full of □'s is considered full.)

  We assume that initially all the zones are half-full. We can ensure this by filling half of each zone with ⊡ symbols in the first time we encounter it.

- The total number of non-⊡ symbols in $L_i \cup R_i$ is 2^i. That is, if L_i is full then R_i is empty and vice versa.

- Location 0 always contains a non-⊡ symbol.

The advantage in setting up these zones is that now when performing the shifts, we do not always have to move the entire tape, but by using the "buffer zones" made up of ⊠ symbols, we can restrict ourselves to only using some of the zones. We illustrate this by showing how $\mathcal{U}$ performs a left shift on the first of its parallel tapes (see Figure 1.10):

1. $\mathcal{U}$ finds the smallest i such that R_i is not empty. Note that this is also the smallest i such that L_i is not full.

2. $\mathcal{U}$ puts the leftmost non-⊠ symbol of R_i in position 0 and shifts the remaining leftmost $2^{i-1}-1$ non-⊠ symbols from R_i into the zones $R_0, \ldots, R_{i-1}$ filling up exactly half the symbols of each zone. Note that there is room to perform this since all the zones $R_0, \ldots, R_{i-1}$ were empty and that indeed $2^{i-1} = \sum_{j=0}^{i-2} 2^j + 1$.

3. $\mathcal{U}$ performs the symmetric operation to the left of position 0: it shifts into L_i the 2^{i-1} leftmost symbols in the zones $L_{i-1}, \ldots, L_1$ and reorganizes $L_{i-1}, \ldots, L_i$ such that the remaining $\sum_{j=1}^{i-1} 2^j - 2^{i-1} = 2^{i-1} - 1$ symbols, plus the symbol that was originally in position 0 (modified appropriately according to M's transition function) take up exactly half of each of the zones $L_{i-1}, \ldots, L_i$.

4. Note that at the end of the shift, all of the zones $L_0, R_0, \ldots, L_{i-1}, R_{i-1}$ are half-full and so we haven't violated our invariant.

Performing such a shift costs $O(\sum_{j=1}^{i} 2^j) = O(2^i)$ operations. However, once we do this, we will not touch L_i again until we perform at least 2^{i-1} shifts (since now the zones $L_0, R_0, \ldots, L_{i-1}, R_{i-1}$ are half-full). Thus, when simulating T steps of M, we perform a shift involving L_i and R_i during the simulation of at most a $\frac{1}{2^{i-1}}$ fraction of these steps. Thus, the total number of operations used by these shifts is when simulating T steps is

$$O(\sum_{i=1}^{\log T+1} \frac{T}{2^{i-1}} 2^i) = O(T \log T) .$$

■