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 problem of mathematically modeling computation may at first seem insurmountable: throughout history people have been solving computational tasks using a wide variety of methods, ranging from intuition and "eureka" moments, to mechanical devices such as abacus or sliderules, to modern computers. Besides that, other organisms and systems in nature are also faced with and solve computational tasks every day using a bewildering array of mechanisms. How can you find a simple mathematical model that captures all of these ways to compute? The problem is even more exacerbated since in this book we are interested in issues of *computational efficiency.* 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 enough, it turns out there there is a simple mathematical model that suffices for studying many questions about computation and its efficiency —the *Turing machine.* It suffices to restrict attention to this single model since it seems able to *simulate* all physically realizable computational methods with little loss of efficiency. Thus the set of "efficiently solvable" computational tasks is at least as large for the Turing Machine as for any other method of computation. (One possible exception is the quantum computer model described in Chapter 10, but we do not currently know if it is physically realizable.)

In this chapter we formally define Turing machines and survey some of their basic properties. Section 1.1 sketches the model and its basic properties. That section also gives an overview of the results of Sections 1.2 to 1.5 for the casual readers who wish to skip the somewhat messy details of the model and go on to complexity theory, which begins with Section 1.6.

Since complexity theory is concerned with *computational efficiency*, Section 1.6 contains one of the most important definitions in this book: the definition of complexity class **P**, which aims to capture mathematically the set of all decision problems that can be efficiently solved. Section 1.6 also contains some discussion on whether or not the class **P** truly captures the informal notion of "efficient computation". 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 machine computations.

1.1 Modeling computation: What you really need to know

Some tedious notation is unavoidable if one talks formally about Turing machines. We provide an intuitive overview of this material for casual readers who can then skip ahead to complexity questions, which begin with Section 1.6. Such a reader can always return to the skipped sections on the rare occasions in the rest of the book when we actually use details of the Turing machine model.

For thousands of years, the term "computation" was understood to mean application of 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 is a concrete embodiment of this intuitive notion. Section 1.2.1 shows that it can be also viewed as the equivalent of any modern programming language — albeit one with no built-in prohibition of its memory size.[1]

Here we describe this model informally along the lines of Turing's quote at the start of the chapter. Let f be a function that takes a string of bits (i.e., a member of the set $\{0,1\}^*$) and outputs, either 0 or 1. 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 $\{0,\dots,9\}$) 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. We measure it in asymptotic terms, so we say a machine runs in time $T(n)$ if it performs at most $T(n)$ basic operations time on inputs of length n.

The following are simple facts about this model.

1. The model is robust to almost any tweak in the definition such as changing the alphabet from $\{0,1,\dots,9\}$ to $\{0,1\}$, or allowing multiple scratchpads, and so on. The most basic version of the model can *simulate* the most complicated version with at most

[1] Though the assumption of a potentially infinite memory may seem unrealistic at first, in the complexity setting it is of no consequence since we will restrict our study to Machines that use at most a finite number of computational steps and memory cells any given input (the number allowed will depend upon the input size).

polynomial (actually quadratic) slowdown. Thus t steps on the complicated model can be simulated in $O(t^c)$ steps on the weaker model where c is a constant depending only on the two models. See Section 1.3.

2. An algorithm (i.e., a machine) can be represented as a bit string once we decide on some canonical encoding. Thus an algorithm/machine can be viewed as a possible *input* to another algorithm —this makes the boundary between *input*, *software* and *hardware* very fluid. (As an aside we note that this fluidity is the basis of a lot of computer technology.) We denote by M_α the machine whose representation as a bit string is α.

3. There is a *universal* Turing machine $\mathcal{U}$ that can *simulate* any other Turing machine given its bit representation. Given a pair of bit strings (x, α) as input, this machine simulates the behavior of M_α on input x. This simulation is very efficient: if the running time of M_α was $T(|x|)$ then the running time of $\mathcal{U}$ is $O(T(|x|)\log T(|x|))$. See Section 1.4.

4. The previous two facts can be used to easily prove the existence of functions that are not computable by any Turing machine; see Section 1.5. Uncomputability has an intimate connection to Gödel's famous Incompleteness Theorem; see Section 1.5.2.

1.2 The Turing Machine

The *k-tape Turing machine* (TM) concretely realizes the above informal notion in the following way (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 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.

There also are variants of Turing machines with *random access memory*,[2] but it turns out that their computational powers are equivalent to standard Turing machines (see Exercise 1.9).

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 (or stay in place).

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

[2] *Random access* denotes the ability to access the i^{th} symbol of the memory within a single step, without having to move a head all the way to the i^{th} location. The name "random access" is somewhat unfortunate since this concept involves no notion of randomness— perhaps "indexed access" would have been better. However, "random access" is widely used and so we follow this convention this book.

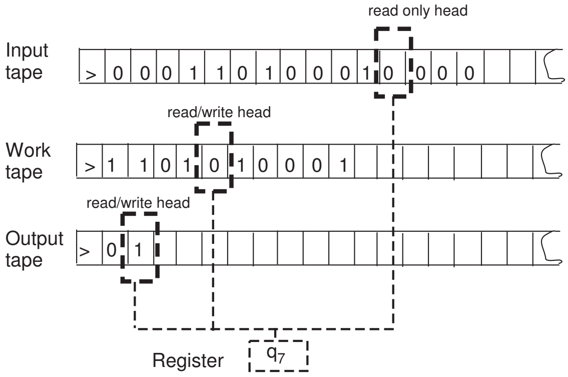


Figure 1.1 A snapshot of the execution of a 3-tape Turing machine M with an input tape, a work tape, and an output tape.

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:

- A finite set Γ of the symbols that M's tapes can contain. We assume that Γ contains a designated "blank" symbol, denoted $\Box$, a designated "start" symbol, denoted $\rhd$ and the numbers 0 and 1. We call Γ the *alphabet* of M.
- A finite 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 $\delta : Q \times \Gamma^k \rightarrow Q \times \Gamma^{k-1} \times \{\mathsf{L}, \mathsf{S}, \mathsf{R}\}^k$, where $k \geq 2$, describing the rules M use 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	**R**ight →	b'	**L**eft ←	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 $\delta(q, (\sigma_1, \ldots, \sigma_k)) = (q', (\sigma'_2, \ldots, \sigma'_k), z)$ where $z \in \{\mathsf{L}, \mathsf{S}, \mathsf{R}\}^k$, 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 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 x ("the input"), and the the blank symbol $\square$ on the rest of its cells. 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.

Example 1.1

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.

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-tape 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 $\{q_{\mathsf{start}}, q_{\mathsf{copy}}, q_{\mathsf{left}}, q_{\mathsf{test}}, q_{\mathsf{halt}}\}$. 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_{left}.
3. On state q_{left}:
   - 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_{left}.
   - 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 □ and the symbol read from the work-tape is the start symbol ▷ 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}.

Clearly, fully specifying a Turing machine is tedious and not always informative. While it is useful to work out one or two examples for yourself (see Exercise 1.1), in the rest of this book we avoid such overly detailed descriptions and specify TM's in a high level fashion. For readers who know how to write computer programs, Example 1.2 below should convince them that they know (in principle at least) how to design a Turing machine for any computational task for which they know how to write computer programs.

1.2.1 The expressive power of Turing machines

At first sight, it may be unclear that Turing machines do indeed 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 1.1). Having done that, you may be ready for the next example; it outlines how you can translate a program in your favorite programming language into a Turing machine. (The reverse direction also holds: most programming languages can simulate a Turing machine.)

Example 1.2 *(Simulating a general programming language using Turing machines)*
(This example assumes some background in computing.)
We give a hand-wavy proof that for any program written in any of the familiar programming languages such as C or Java, there is an equivalent Turing machine. 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 instructions, each of one of a few simple types, e.g., (a) read from memory into one of a finite number of registers (b) write a register's contents to memory, (c) Add the contents of two registers and store the result in a third. (d) Like (c) but with other operations such as multiplication instead of addition. All these operations can be easily simulated by a Turing machine. The memory and registers 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 design TM's that add or multiply two numbers. To simulate the computer's memory, a two-tape TM can use one tape for the simulated memory and the other tape to do binary-to-unary conversion that allows it, for a number i in binary representation, to read or modify the i^{th} location of its first tape. We leave details to Exercise 1.8.
Exercise 1.10 asks you to give a more rigorous proof of such a simulation for a simple tailor-made programming language.

1.3 Efficiency and running time

Now we formalize the notion of running time. As every non-trivial computational task requires at least reading the entire input, we count the number of basic steps *as a function of the input length.*

Definition 1.3 *(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 if for every $x \in \{0,1\}^*$, whenever M is initialized to the start configuration on input x, then it halts with $f(x)$ written on its output tape. We say M *computes* f *in* $T(n)$*-time*[3] if its computation on every input x requires at most $T(|x|)$ steps.

Example 1.4
It is easily checked that the Turing machine for palindrome recognition in Example 1.1 runs in $3n$ time.

Time-constructible functions. 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 we will restrict our attention to time bounds of this form. (Allowing time bounds that are not time-constructible can lead to anomalous results.) The restriction $T(n) \geq n$ is to allow the algorithm time to read its input.

1.3.1 Robustness of our definition

Most of the specific details of our definition of Turing machines are quite arbitrary. It is a simple exercise to see that most changes to the definition do not yield a substantially different model, since our model can simulate any of these new models. In context of computational complexity, however, we have to verify not only that one model can simulate another, but that it can do so efficiently. Now we state a few results of this type, which ultimately lead to the conclusion that the exact model is unimportant if we are willing to ignore polynomial factors in the running time. Variations on the model studied include restricting the alphabet Γ to be $\{0,1,\Box,\rhd\}$, restricting the machine to have a single work tape, or allowing the tapes to be infinite in both directions. All results in this section are proved sketchily— completing these sketches into full proofs is a very good way to gain intuition on Turing machines, see Exercises 1.2, 1.3 and 1.4.

Claim 1.5 *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* $\{0,1,\Box,\rhd\}$. $\diamondsuit$

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 describe an equivalent TM $\tilde{M}$ computing f with alphabet $\{0,1,\Box,\rhd\}$, k tapes and a set Q' of states. The idea behind the transformation is simple: one can encode any member of Γ using $\log |\Gamma|$ bits.[4] Thus, each of $\tilde{M}$'s work tapes will

[3] 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.

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

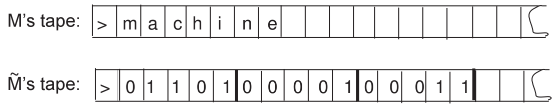


Figure 1.3 We can simulate a machine M using the alphabet $\{\triangleright, \square, \mathtt{a}, \mathtt{b}, \ldots, \mathtt{z}\}$ by a machine M' using $\{\triangleright, \square, 0, 1\}$ via encoding every tape cell of M using 5 cells of M'.

simply encode one 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, **(4)** store this information in its state register, and **(5)** 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 $\log|\Gamma|$. Thus, there is such a machine $\tilde{M}$ utilizing no more than $c|Q||\Gamma|^{k+1}$ states for some absolute constant c. (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. ■

Now we consider the effect of restricting the machine to use a *single tape*— one read/write tape that serves as input, work and output tape (this is the standard computational model in many undergraduate texts such as [Sip96]). We show that going from multiple tapes to a single tape can at most square the running time. This quadratic increase is inherent for some languages, including the palindrome recognition considered in Example 1.1; see the chapter notes.

Claim 1.6 *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. 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 k tapes then it is computable in time $5kT(n)^2$ by a single-tape TM $\tilde{M}$.* ♢

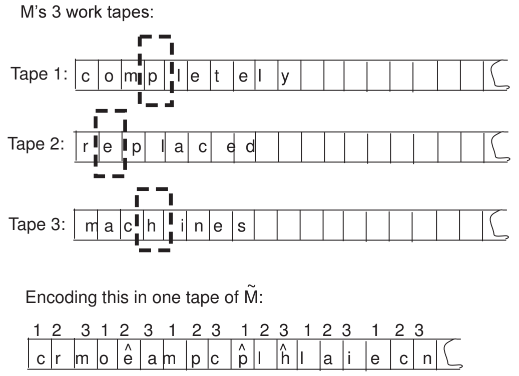


Figure 1.4 Simulating a machine M with 3 tapes using a machine $\tilde{M}$ with a single tape.

Proof Sketch: Again the idea is simple: the TM $\tilde{M}$ encodes 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 "ˆ type", indicating that the corresponding head of M is positioned in that location (see figure). $\tilde{M}$ will not touch the first $n+1$ locations of its tape (where the input is located), but rather start by taking $O(n^2)$ steps to copy the input bit by bit into the rest of the tape, while encoding it in the above way.

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 $2n + kT(n) \leq (k+2)T(n)$ of its work tape, meaning that for each of the at most $T(n)$ steps of M, $\tilde{M}$ performs at most $5 \cdot k \cdot T(n)$ work (sweeping back and forth requires about $4 \cdot k \cdot T(n)$ steps, and some additional steps may be needed for updating head movement and book keeping). ■

Remark 1.7 *(Oblivious Turing machines.)*
With a bit of care, one can ensure that the proof of Claim 1.6 yields a TM $\tilde{M}$ with the following property: its head movements do not depend on the input but only depend on the input length. That is, 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. A machine with this property is called *oblivious* and the fact that every TM can be simulated by an oblivious TM will simplify some proofs later on (see Exercises 1.5, 1.6 and the proof of Theorem 2.10).

Claim 1.8 *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 $\tilde{M}$.* ♢

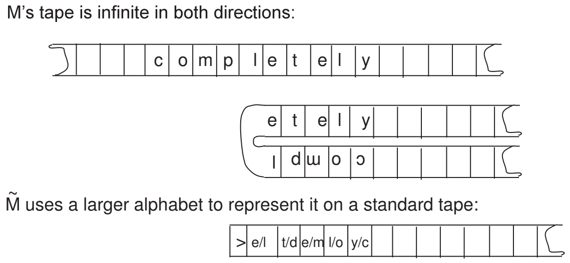

M's tape is infinite in both directions:
c o m p l e t e l y
e t e l y
l d m o c
M̃ uses a larger alphabet to represent it on a standard tape:
> e/l t/d e/m l/o y/c

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 do 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 1.7 and 1.9; also the texts [Sip96, HMU01] contain more examples). In particular none of these modifications will change the class **P** of polynomial-time computable decision problems defined below in Section 1.6.

1.4 Machines as strings and the universal Turing machine

It is almost obvious that we can represent a Turing machine as a string: just write the description of the TM on paper, and encode this description as a sequence of zeros and ones. This string can be given as input to another TM. This simple observation is actually profound since it blurs the distinction between *software*, *hardware* and *data*. Historically speaking it motivated the invention of the *general purpose* electronic computer, which is a single machine that can be adapted to any arbitrary task by loading it with an appropriate program (software).

Because we will use this notion of representing TM's as strings quite extensively, it may be worthwhile to spell out our representation 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.[5] 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 a somewhat similar effect to the *comments* mechanism of many programming languages (e.g., the `/*...*/` construct in C,C++ and Java) that allows to add superfluous symbols to any program.

We denote by $\llcorner M \lrcorner$ the TM M's representation as a binary string. If α is a string then M_α denotes the TM that α represents. As is our convention, we will also often use M to denote both the TM and its representation as a string. Exercise 1.11 asks you to fully specify a representation scheme for Turing machines with the above properties.

1.4.1 The Universal Turing Machine

Turing was the first to observe 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, this suffices to represent the other machine's state 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 [HS66]. 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).

[5] 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 $\rhd, \Box, 0, 1$ are the first 4 symbols of the machine's alphabet.

Theorem 1.9 *(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.

Moreover, 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.

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 1.9 can be considered a variant of this exercise.

Proof of relaxed version of Theorem 1.9: 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 1.5 and 1.6. 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).

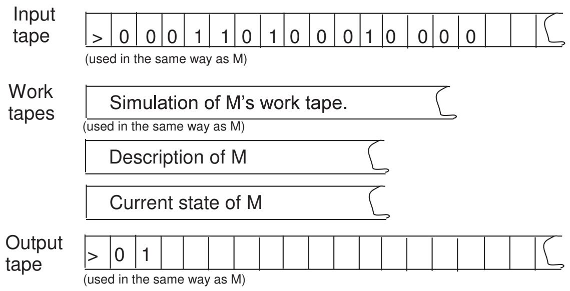

Input tape
> 0 0 0 1 1 0 1 0 0 0 1 0 0 0 0
(used in the same way as M)
Work tapes
Simulation of M's work tape.
(used in the same way as M)
Description of M
Current state of M
Output tape
> 0 1
(used in the same way as M)

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 1.5 and 1.6 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 be turned into an exact specification of the TM $\mathcal{U}$, though we leave this to the reader. To work out the details, it may help to think first how to program these steps in your favorite programming language and then try to transform this into a description of a Turing machine. ■

Universal TM with time bound. 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 time counter to $\mathcal{U}$, the proof of Theorem 1.9 can be easily modified to give such a universal TM. The time counter is used to keep track of the number of steps that the computation has taken so far.

1.5 Uncomputability: An introduction

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! This section gives a brief introduction to this fact and its ramifications. Though not this material is not strictly necessary for the study of complexity, it forms the intellectual background for it.

The next theorem shows the existence of uncomputable functions. In fact it shows the existence of such functions whose range is $\{0,1\}$, in other words, they represent *languages*. Such a language is called *undecidable*. The theorem's proof uses a technique called *diagonalization*, which is useful in complexity theory as well; see Chapter 3.

Theorem 1.10 *There exists a function* $\mathsf{UC} : \{0,1\}^* \to \{0,1\}$ *that is not computable by any TM.* ♢

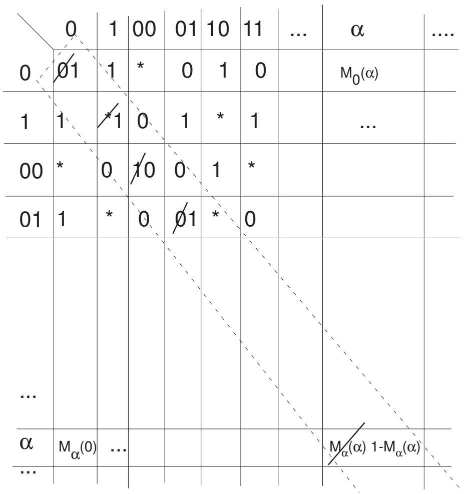
Proof: The function UC is defined as follows: for every $\alpha \in \{0,1\}^*$, if $M_\alpha(\alpha) = 1$ then $\mathsf{UC}(\alpha) = 0$; otherwise (if $M_\alpha(\alpha)$ outputs a different value or enters an infinite loop), $\mathsf{UC}(\alpha) = 1$.

Suppose for the sake of contradiction that UC is computable and hence 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,

$$\mathsf{UC}(\llcorner M \lrcorner) = 1 \Leftrightarrow M(\llcorner M \lrcorner) \neq 1 \,.$$

■

To see why this proof technique is called "diagnalization," see Figure 1.7.



	0	1	00	01	10	11	...	α	....
0	~~0~~1	1	*	0	1	0		$M_0(\alpha)$	
1	1	~~*~~1	0	1	*	1		...	
00	*	0	~~1~~0	0	1	*			
01	1	*	0	~~0~~1	*	0			
...									
α	$M_\alpha(0)$	...						~~$M_\alpha(\alpha)$~~ $1\text{-}M_\alpha(\alpha)$	
...									

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, function UC is defined by "negating" the diagonal of this table. Since the rows of the table represent *all* TMs, we conclude that UC cannot be computed by any TM.

1.5.1 The Halting Problem (first encounter with reductions)

The reader may well ask why should we care whether or not the function UC described above is computable— who would want to compute such a contrived function anyway? We now show a more natural uncomputable function. The function HALT takes as input a pair $\langle \alpha, x \rangle$ 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. If computers could compute HALT, the task of designing bug-free software and hardware would become much easier. Unfortunately, we now show that computers cannot do this, even if they are allowed to run an arbitrarily long time:

Theorem 1.11 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 1.10.

The TM M_{UC} is simple: on input α, M_{UC} runs $M_{\mathsf{HALT}}(\alpha, \alpha)$. If the result is 0 (meaning that M_α does not halt on α) then M_{UC} outputs 1. Otherwise, M_{UC} uses the universal TM $\mathcal{U}$ to compute $b = M_\alpha(\alpha)$. If $b = 1$ then M_{UC} outputs 0; otherwise it outputs 1.

Under the assumption that $M_{\mathsf{HALT}}(\alpha, \alpha)$ outputs $\mathsf{HALT}(\alpha, \alpha)$ within a finite number of steps, the TM $M_{\mathsf{UC}}(\alpha)$ will output $\mathsf{UC}(\alpha)$. ■

The proof technique employed to show Theorem 1.11 is called a *reduction*. We showed that computing UC is *reducible* to computing HALT —we showed that if there were a hypothetical algorithm for HALT then there would be one for UC. 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 of interesting uncomputable (also known as *undecidable*) functions, see Exercise 1.12. 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 an algorithm for solving it (which he presumed to exist) as one of the top 23 open problems in mathematics. The chapter notes mention some good sources for more information on computability theory.

1.5.2 Gödel's Theorem

In the year 1900, David Hilbert, the preeminent mathematician of his time, proposed an ambitious agenda to base all of mathematics on solid axiomatic foundations, so that eventually all true statements would be rigorously proven. Mathematicians such as Russell, Whitehead, Zermelo, and Fraenkel, proposed axiomatic systems in the ensuing decades, but nobody was able to prove that their systems are simultaneously *complete* (i.e., prove all true mathematical statements) and *sound* (i.e., prove no false statements). In 1931 Kurt Gödel shocked the mathematical world by showing that this ongoing effort is doomed to fail —for every sound system $\mathcal{S}$ of axioms and rules of inference, there exist true number theoretic statements that cannot be proven in $\mathcal{S}$.

Gödel's work directly motivated the work of Turing and Church on computability. Our presentation reverses this order: we use uncomputability to sketch a proof of Gödel's result. The main observation is the following: in any sufficiently powerful axiomatic system, for any input $\langle \alpha, x \rangle$ we can write a mathematical statement $\phi_{\langle \alpha, x \rangle}$ that is true iff $\mathsf{HALT}(\langle \alpha, x \rangle) = 1$. (A sketch of this construction appears below.) Now if the system is complete, it must prove at least one of $\phi_{\langle \alpha, x \rangle}$ or $\neg\phi_{\langle \alpha, x \rangle}$, and if it is sound it cannot prove both. So if the system is both complete and sound, the following algorithm for the Halting problem is guaranteed to terminate in finite time for all inputs. "*Given input $\langle \alpha, x \rangle$, start enumerating all strings* *of finite length, and check for each generated string whether it represents a proof in the axiomatic system for either $\phi_{\langle\alpha,x\rangle}$ or $\neg\phi_{\langle\alpha,x\rangle}$. If one waits long enough, a proof of one of the two statements will appear in the enumeration, at which point the correct answer* 1 *or* 0 *is revealed, which you then output.*" (Note that this procedure implicitly uses the simple fact that proofs in axiomatic systems can be easily verified by a Turing machine, since each step in the proof has to follow mechanically from previous steps by applying the axioms.)

Now we sketch the construction of the desired statement $\phi_{\langle\alpha,x\rangle}$. Assume the axiomatic system has the ability to express statements about the natural number using the operators plus ($+$) and times ($\times$), equality and comparison relations ($=, >, <$), and logical operators such as AND ($\wedge$), OR ($\vee$), and NOT ($\neg$). The language also includes the quantifiers for-all ($\forall$) and exists ($\exists$) and the constant 1 (we can get any other constant c by adding 1 to itself c times). For example, the formal expression for "x divides y" will be $\mathsf{DIVIDES}(x,y) = \exists_k \;:\; y = x \times k$, and the expression for "y is prime" will be $\mathsf{PRIME}(y) = \forall_x (x=1) \vee (x=y) \vee \neg\mathsf{DIVIDES}(x,y)$ (where $\mathsf{DIVIDES}(x,y)$ is shorthand for the corresponding expression).

We can encode strings (and hence also Turing machines and their inputs and tapes) as numbers. Then one notes that a basic operation of the Turing machine only influences one (or a few, if the machine has mutiple tapes) of bits on its tape, which can be viewed as a simple arithmetic operation on the string/number representing the tape contents. With some work one obtains an expression $\varphi_{\alpha,x}(t)$ that is true if and only if the TM M_α halts on input x within t steps. Hence, M_α halts on x if and only if $\exists_t \varphi_{\alpha,x}(t)$ is true, which is the desired mathematical statement. We leave the details as Exercise 1.13.

Note that this construction also implies, as first pointed out by Turing, that the set of true mathematical statements is undecidable, which showed that Hilbert's famous *Entscheidungsproblem* has no solution. (Hilbert had asked for a "mechanical procedure" —now interpreted as "algorithmic procedure"— for deciding truth of mathematical statements.)

1.6 The class $\mathbf{P}$

A *complexity class* is a set of functions that can be computed within given resource bounds. We will now introduce our first complexity class. For reasons of technical convenience, throughout most of this book we will pay special attention to Boolean functions, namely those that have only one bit of output. These functions define *decision problems* or *languages*. We say that a machine *decides* a language $L \subseteq \{0,1\}^*$ if it computes the function $f_L : \{0,1\}^* \to \{0,1\}$ where $f_L(x) = 1 \Leftrightarrow x \in L$.

Definition 1.12 *(The class* **DTIME***.)* Let $T : \mathbb{N} \to \mathbb{N}$ be some function. A language L is in $\mathbf{DTIME}(T(n))$ iff there is a Turing machine that runs in time $c \cdot T(n)$ for some constant $c > 0$ and decides L. ♢

The "D" in the notation **DTIME** refers to "deterministic". The Turing machine introduced in this chapter is more precisely called the *deterministic* Turing machine since for any given input x, the machine's computation can proceed in exactly one way. Later we will see other types of Turing machines, including nondeterministic and probabilistic TMs.

Now we try to make the notion of "efficient computation" precise. We equate this with *polynomial* running time, which means it is at most n^c for some constant $c > 0$. The following class captures this notion, where **P** stands for "polynomial."

Definition 1.13 *(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 0.1.

Example 1.14 *(Graph Connectivity)*
In the *graph connectivity* problem, we are given a graph G and two vertices s, t in G. We have to decide if s is connected to t in G. This problem is in $\mathbf{P}$. The algorithm that shows this uses *depth-first search*, a simple idea taught in undergraduate courses. The algorithm explored the graph edge-by-edge starting from s, marking visited edges. In subsequent edges it also tries to explore all unvisited edges that are adjacent to previously-visited edges. After at most $\binom{n}{2}$ steps, all edges are either visited or will never be visited.
See Exercise 1.14 for more examples of languages in $\mathbf{P}$.

Example 1.15
We give some examples to emphasize a couple of points about the definition of the class $\mathbf{P}$. First, the class contains only decision problems. Thus we cannot say, for example, that "Integer multiplication is in $\mathbf{P}$." Instead, we may say that its decision version is in $\mathbf{P}$, namely, the following language:

$$\left\{ \langle x, i \rangle : \text{The } i^{th} \text{ bit of } xy \text{ is equal to } 1 \right\} .$$

Second, the running time is a function of the number of *bits* in the input. Consider the problem of solving a system of linear equations over the rational numbers. In other words, given a pair $\langle A, \mathbf{b} \rangle$ where A is an $m \times n$ rational matrix and $\mathbf{b}$ is an m dimensional rational vector, find out if there exists an n-dimensional vector $\mathbf{x}$ such that $A\mathbf{x} = \mathbf{b}$. The standard Gaussian Elimination algorithm solves this problem in $O(n^3)$ *arithmetic operations*. But on a Turing machine, each arithmetic operation has to be done in the gradeschool fashion, bit by laborious bit. Thus to prove that this decision problem is in $\mathbf{P}$ we have to verify that Gaussian elimination (or some other algorithm for the problem) runs on a Turing machine in time that is polynomial in the number of bits required to represent $a_1, a_2, \ldots, a_n$. That is, in the case of Gaussian elimination, we need to verify that all the intermediate numbers involved in the computation can be represented by polynomially many bits. Fortunately, this does turn out to be the case (for a related result, see Exercise 2.3).

1.6.1 Why the model may not matter

We defined the classes of "computable" languages and $\mathbf{P}$ using Turing machines. Would they be different if we had used a different computational model? Would these classes be different for some an advanced alien civilization, which has discovered computation but with a different computational model than the Turing machine?

We already encountered variations on the Turing machine model, and saw that the weakest one can simulate the strongest one with quadratic slow down. Thus *polynomial* time is the same on all these variants, as is the set of computable problems.

In the few decades after Church and Turing's work many other models of computation were discovered, some quite bizarre. It was easily shown that the Turing machine can simulate all of them with at most polynomial slowdown. Thus the analogue of $\mathbf{P}$ on these models is no larger than that for the Turing machine.

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. This implies that the set of *computable* problems would be no larger on any other computational model that on the Turing machine. (The CT thesis is not a theorem, merely a belief about the nature of the world as we currently understand it.)

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, this strong form is somewhat controversial, in particular because of models such as *quantum computers* (see Chapter 10), which do not appear to be efficiently simulatable on TMs. However, it is still unclear if quantum computers can be physically realized.

1.6.2 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 since n^{100} is prohibitively huge even for moderate values of n. 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} time 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 time algorithm.)

Note that the class **P** is useful only in a certain context. Turing machines are a crude 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, the language INDSET of Example 0.1, then even an n^{20} time algorithm would be a fantastic breakthrough.

P is also a natural class from the viewpoint of a programmer. Suppose a programmer is asked to invent the definition of an "efficient" computation. Presumably, she 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), she might find it natural to consider a program "efficient" if it performs only "efficient" computations and calls subroutines that are "efficient". The resulting notion of "efficient computations" obtained turns out to be exactly the class **P** [Cob64].

1.6.3 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 on *every* possible input. Critics point out that not all possible inputs arise in practice, and our algorithms only need to be efficient on the types of inputs that do arise. This criticism is partly answered using *average-case complexity* and by defining an analog of **P** in that context; see Chapter 18. We also note that quantifying "real life" distributions is tricky.

Similarly, in context of computing functions such as the size of the largest independent set in the graph, users are often willing to settle for *approximate* solutions. Chapters 11 and 22 contain a rigorous treatment of the complexity of approximation.

Other physically realizable models. We already mentioned the strong form of the Church Turing thesis, which posits that the class **P** is not any larger for any physically realizable computational model. However, some subtleties need discusssion.

(a) *Issue of precision:* TM's compute with discrete symbols, whereas physical quantities may be real numbers in $\mathbb{R}$. Thus one can conceive of computational models based upon physics phenomena that may be able to operate over real numbers. Because of the precision issue, a TM can only approximately simulate such computations. It seems though that TMs do not suffer from an inherent handicap (though a few researchers disagree). After all, real-life devices suffer from noise, and physical quantities can only be measured up to finite precision. Thus physical processes could not involve arbitrary precision, and the simulating TM can therefore simulate them using finite precision.

Even so, in Chapter 16 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 standard 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 complexity class **BPP**, which is the analogue of **P** for those machines. However, we will see in Chapters 19 and 20 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 10 we define the complexity class **BQP**, that generalizes **P** in such a way. We will see problems in **BQP** that are currently not known to be in **P** (though there is no known proof that $\mathbf{BQP} \neq \mathbf{P}$). However, it is not yet clear whether the quantum model is truly physically realizable. Also quantum computers currently seem only able to efficiently solve only very few problems that are not known to be in **P**. Hence some insights gained from studying **P** may still apply to quantum computers.

(d) *Use of other exotic physics, such as string theory.* So far it seems that many such physical theories yield the same class **BQP**, though much remains to be understood.

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.

1.6.4 Edmonds' quote

We conclude this section with a quote from Edmonds [Edm65], who in his celebrated paper on 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 simulate (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.
- 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 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 [Chu36]. A few months later, Alan Turing independently introduced his Turing machines and showed functions inherently uncomputable by such machines [Tur36]. 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 [Dav65] contains many seminal papers on computability. Part II of Sipser's book [Sip96] is a good gentle introduction to this theory, while the books [Rog87, HMU01, Koz97] go into a bit more depth. These books also cover *automata theory*, which is another area of the theory of computation not discussed in the current book. This book's website contains some additional links for information on both these topics.

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 [Hod83, Lea05]).[6] 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 von Neumann, an extremely prolific scientist that was involved in everything 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 [vN45].

[6] 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.

As computers became more prevalent, the issue of efficiency in computation began to take center stage. Cobham [Cob64] defined the class $\mathbf{P}$ and suggested it may be a good formalization for efficient computation. A similar suggestion was made by Edmonds ([Edm65], 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 [HS65] defined the class $\mathbf{DTIME}(T(n))$ for every function T, and proved the slightly relaxed version of Theorem 1.9 we showed above (the version we stated and prove below was given by Hennie and Stearns [HS66]). 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 [Blu67] has given 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.

The result that PAL requires $\Omega(n^2)$ steps to compute on TM's using a single read/write tape is from [Maa84], see also Exercise 13.3. We have stated that algorithms that take less than n steps are not very interesting as they do not even have time to read their input. This is true for the Turing machine model. However, if one allows *random access* to the input combined with *randomization* then many interesting computational tasks can actually be achieved in *sublinear* time. See [Fis04] for a survey of this line of research.

Exercises

1.1 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. H457

1.2 Complete the proof of Claim 1.5 by writing down explicitly the description of the machine $\tilde{M}$.

1.3 Complete the proof of Claim 1.6.

1.4 Complete the proof of Claim 1.8.

1.5 Define a TM M to be *oblivious* if its head movements do 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. H457

1.6 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 $O(T(n) \log T(n))$. H457

1.7 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 $f \in \mathbf{DTIME}(T(n)^2)$.

1.8 Let LOOKUP denote the following function: on input a pair $\langle x, i\rangle$ (where x is a binary string and i is a natural number), LOOKUP outputs the i^{th} bit of x or 0 if $|x| < i$. Prove that LOOKUP $\in \mathbf{P}$.

1.9 Define a *RAM Turing machine* to be a Turing machine that has *random access memory.* We formalize this as follows: the machine has an infinite array A that is initialized to all blanks. It accesses this array as follows. One of the machine's work tapes is designated as the *address tape.* Also the machine has two special alphabet symbols denoted by `R` and `W` and an additional state we denote by q_{access}. 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 $\mathbf{DTIME}(T(n)^2)$.

1.10 Consider the following simple programming language. It has a single infinite array `A` of elements in $\{0,1,\Box\}$ (initialized to $\Box$) 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 $\sigma \in \{0,1,\Box\}$ and *cmds* is a list of one or more of the following commands: **(1)** `Set A[i] to` τ where $\tau \in \{0,1,\Box\}$, **(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 $f \in \mathbf{DTIME}(T(n))$.

1.11 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 1.1) 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).

1.12 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 1.11 by showing that the function HALT is not computable.

(b) Prove Rice's Theorem. H457

1.13 It is known that there is some constant C such that for every $i > C$ there is a prime larger than i^3 but smaller than $(i+1)^3$ [Hoh30, Ing37]. For every $i \in \mathbb{N}$, let p_i denote the smallest prime between $(i+C)^3$ and $(i+C+1)^3$. We say that a number n encodes a string $x \in \{0,1\}^*$, if for every $i \in \{1..|x|\}$, p_i divides n if and only if $x_i = 1$.[7]

(a) Show a logical expression $\mathsf{BIT}(n,i)$ that is true if and only if p_i divides n.

(b) Show a logical expression $\mathsf{COMPARE}(n,m,i,j)$ that is true if and only if the strings encoded by the numbers n and m agree between the i^{th} and j^{th} position.

(c) A *configuration* of a TM M is the contents of all its input tapes, its head location and the state of its register. That is, it contains all the information about M at a particular moment in its execution. Show that such a configuration can be represented by a binary string. (You may assume that M is a single-tape TM as in Claim 1.6.)

(d) For a TM M and input $x \in \{0,1\}^*$, show an expression $\mathsf{INIT}_{M,x}(n)$ that is true if and only if n encodes the initial configuration of M on input x.

(e) For a TM M show an expression $\mathsf{HALT}_M(n)$ that is true if and only if n encodes a configuration of M after which M will halt its execution.

(f) For a TM M, show an expression $\mathsf{NEXT}(n,m)$ that is true if and only if n,m encode configurations x,y of M such that y is the configuration that is obtained from x by a single computational step of M.

(g) For a TM M, show an expression $\mathsf{VALID}_M(m,t)$ that is true if and only m a tuple of t configurations $x_1,\ldots,x_t$ such that x_{i+1} is the configuration obtained from x_i in one computational step of M.

(h) For a TM M and input $x \in \{0,1\}^*$, show an expression $\mathsf{HALT}_{M,x}(t)$ that is true if and only if M halts on input x within t steps.

(i) Let TRUE-EXP denote the function that on input (a string representation of) a number-theoretic statement φ (composed in the formalism above), outputs 1 if φ is true, and 0 if φ is false. Prove that TRUE-EXP is uncomputable.

1.14 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.

[7] Technically speaking under this definition a number can encode more than one string. This will not be an issue, though we can avoid it by first encoding the string x as a $2|x|$ bit string y using the map $0 \mapsto 00, 1 \mapsto 11$ and then adding the sequence 01 at the end of y.

(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).

1.15 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 $d \in \{0..d-1\}$ by its binary representation. The *unary* representation of n, denoted by $\llcorner n \lrcorner_{\mathrm{unary}}$ 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 $\mathbf{P}$. That is, show that for every subset S of the natural numbers, if we define $L_S^b = \{ \llcorner n \lrcorner_b : n \in S\}$ then for every $b \geq 2$, $L_S^b \in \mathbf{P}$ iff $L_S^2 \in \mathbf{P}$.

(b) Show that choosing the unary representation may make a difference by showing that the following language is in $\mathbf{P}$:

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

It is not known to be in $\mathbf{P}$ if we choose the binary representation (see Chapters 9 and 10). In Chapter 3 we will see that there is a problem that is *proven* to be in $\mathbf{P}$ when choosing the unary representation but not in $\mathbf{P}$ when using the binary representation.

1.A Proof of Theorem 1.9: Universal Simulation in $O(T \log T)$-time

We now show how to prove Theorem 1.9 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 1.4.1. $\mathcal{U}$ will use its input and output tape in the same way M does, and will also have extra work tapes to store M's transition table and current state, and to encode the contents of M's work tapes. The main obstacle we need to overcome is that we cannot use Claim 1.6 to reduce the number of M's work tapes to one, since Claim 1.6 introduces too much overhead in the simulation. Therefore, we will show a different way to encode all of M's work tapes in a single tape of $\mathcal{U}$, which we call the *main* work tape of $\mathcal{U}$.

Let k be the number of tapes that M uses (apart from its input and output tapes) and Γ its alphabet. Following the proof of Claim 1.5, we may assume that $\mathcal{U}$ uses the alphabet Γ^k (as this can be simulated with a overhead depending only on $k, |\Gamma|$). Thus we can encode in each cell of $\mathcal{U}$'s main work tape k symbols of Γ, each corresponding to a symbol from one of M's tapes. This means that we can think of $\mathcal{U}$'s main work tape not as a single tape but rather as *k parallel tapes*; that is, we can think of $\mathcal{U}$ as having k tapes with the property that in each step all their read/write heads go in unison either one location to the left, one location to the right or they all stay in place. While we can easily encode the contents of M's k work tapes in $\mathcal{U}$'s k parallel tapes, we still have to deal with the fact that M's k read/write heads can each move independently to the left or right, whereas $\mathcal{U}$'s parallel tapes are forced to move together. Paraphrasing the famous saying, our strategy to handle this is: *"If the head cannot go to the tape locations then the locations will go to the head"*.

That is, since we can not move $\mathcal{U}$'s read/write head in different directions at once, we simply move the parallel tapes "under" the head. 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 (see Figure 1.8). $\mathcal{U}$ can easily perform such shifts using an additional "scratch" work tape.

The approach above is still not good enough to get $O(T \log T)$-time simulation. The reason is that there may be as many as T non-blank symbols in each parallel tape, and so each shift operation may cost $\mathcal{U}$ as much as T operations per each step of M, resulting in $\Theta(T^2)$-time simulation. We will deal with this problem by encoding the information on the tapes in a way that allows us to amortize the work of performing a shift. We will ensure that we do not need to move the entire non-blank symbols of the tape in each shift operation. Specifically, we will encode the information in a way that allows half of the shift operations to be performed using $2c$ steps, for some constant c, a quarter of them using $4c$ steps, and more generally 2^{-i} fraction of the operations will take $2^i c$ steps, leading to simulation in roughly $cT \log T$ time (see below). (This kind of analysis is called *amortized analysis* and is widely used in algorithm design.)

Encoding M's tapes on $\mathcal{U}$'s tape. To allow more efficient shifts we encode the information using "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 1.8). 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 move it temporarily to perform a shift when, following our

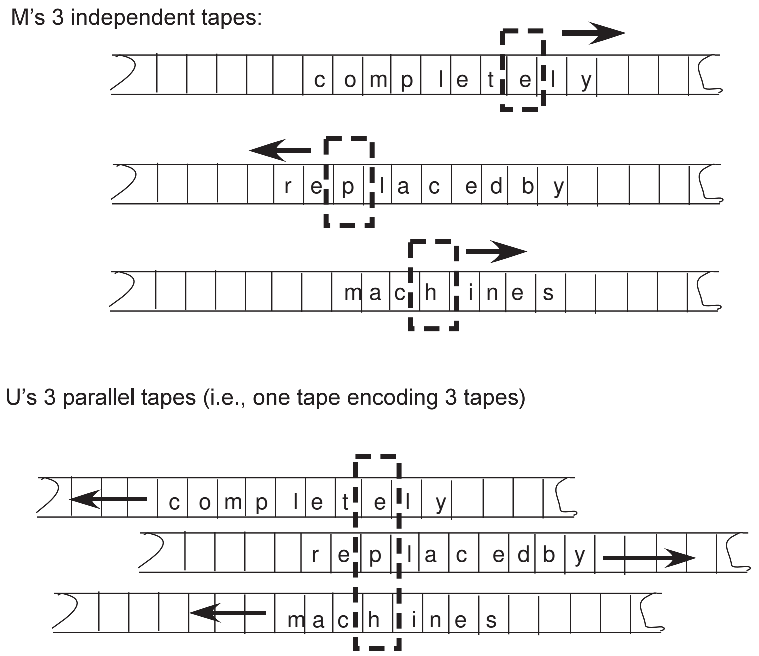


Figure 1.8 Packing k tapes of M into one tape of $\mathcal{U}$. We consider $\mathcal{U}$'s single work tape to be composed of k parallel tapes, whose heads move in unison, and hence we shift the contents of these tapes to simulate independent head movement.

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 each of $\mathcal{U}$'s parallel tapes into *zones* that we denote by $R_0, L_0, R_1, L_1, \ldots$ (we'll only need to go up to $R_{\log T}, L_{\log T}$). The cell at location 0 is not at any zone. Zone R_0 contains the two cells immediately to the right of location C (i.e., locations $+1$ and $+2$), while Zone R_1 contains the four cells $+3, +4, +5, +6$. Generally, for every $i \geq 1$, Zone R_i contains the $2 \cdot 2^i$ cells that are to the right of Zone R_{i-1} (i.e., locations $[2^{i+1}-1..2^{i+2}-2]$). Similarly, Zone L_0 contains the two cells indexed by -1 and -2, and generally Zone L_i contains the cells $[-2^{i+2}+2..-2^{i+1}+1]$. 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 R_i that are not ⊠ is either 0,2^i, or $2 \cdot 2^i$ and the same holds for L_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.)

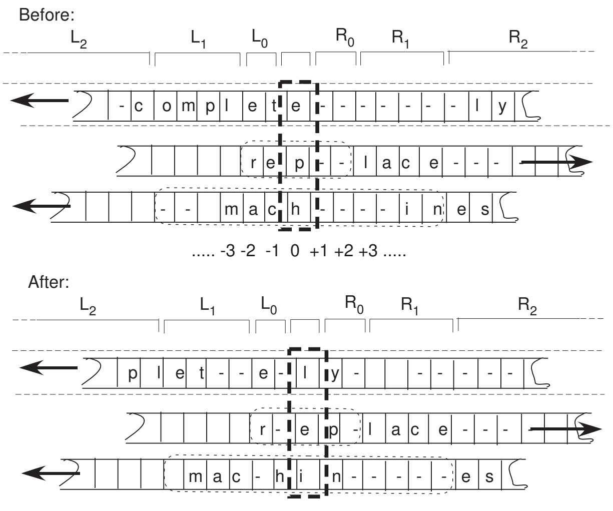
  We assume that initially all the zones are half-full. We can ensure this by filling half of each zone with ⊠ symbols in the first time we encounter it.

- The total number of non-⊠ symbols in $R_i \cup L_i$ is $2 \cdot 2^i$. That is, either R_i is empty and L_i is full, or R_i is full and L_i is empty, or they are both half-full.

- Location 0 always contains a non-⊠ symbol.

Performing a shift. 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 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 also Figure 1.9):

1. $\mathcal{U}$ finds the smallest i_0 such that R_{i_0} is not empty. Note that this is also the smallest i_0 such that L_{i_0} is not full. We call this number i_0 the *index* of this particular shift.


Before:
L2 L1 L0 R0 R1 R2
- c o m p l e t e - - - - - - - - l y
r e p - l a c e - - -
- - m a c h - - - - i n e s
..... -3 -2 -1 0 +1 +2 +3 .....
After:
L2 L1 L0 R0 R1 R2
p l e t - - e - l y - - - - - -
r - e p - l a c e - - -
m a c - h i n - - - - - e s

Zone:	L_i	L_{i-1}	L_{i-2}	...	L_1	L_0	0	R_0	R_1	...	R_{i-2}	R_{i-1}	R_i
no of non-empty locations:	2^i	$2*2^{i-1}$	$2*2^{i-2}$		4	2	1	0	0		0	0	2^i

Zone:	L_i	L_{i-1}	L_{i-2}	...	L_1	L_0	0	R_0	R_1	...	R_{i-2}	R_{i-1}	R_i
no of non-empty locations:	$2*2^i$	2^{i-1}	2^{i-2}		2	1	1	1	2		2^{i-2}	2^{i-1}	0

Figure 1.9 Performing a shift of the parallel tapes. The left shift of the first tape involves zones $R_0, L_0, R_1, L_1, R_2, L_2$, the right shift of the second tape involves only R_0, L_0, while the left shift of the third tape involves zones R_0, L_0, R_1, L_1. We maintain the invariant that each zone is either empty, half-full or full and that the total number of non-empty cells in $R_i \cup L_i$ is $2 \cdot 2^i$. If before the left shift zones $L_0, .., L_{i-1}$ were full and L_i was half-full (and so $R_0, .., R_{i-1}$ were full and R_i half-full), then after the shift zones $R_0, L_0, .., R_{i-1}, L_{i-1}$ will be half-full, L_i will be full and R_i will be empty.

2. $\mathcal{U}$ puts the leftmost non-⊠ symbol of R_{i_0} in position 0 and shifts the remaining leftmost $2^{i_0} - 1$ non-⊠ symbols from R_{i_0} into the zones $R_0, \ldots, R_{i_0-1}$ filling up exactly half the symbols of each zone. Note that there is exactly room to perform this since all the zones $R_0, \ldots, R_{i_0-1}$ were empty and indeed $2^{i_0} - 1 = \sum_{j=0}^{i_0-1} 2^j$.
3. $\mathcal{U}$ performs the symmetric operation to the left of position 0. That is, for j starting from$i_0 - 1$ down to 0, $\mathcal{U}$ iteratively moves the $2 \cdot 2^j$ symbols from L_j to fill half the cells of L_{j+1}. Finally, $\mathcal{U}$ moves the symbol originally in position 0 (modified appropriately according to M's transition function) to L_0.
4. At the end of the shift, all of the zones $R_0, L_0, \ldots, R_{i_0-1}, L_{i_0-1}$ are half-full, R_{i_0} has 2^{i_0} fewer non-⊠ symbols, and L_i has 2^i additional non-⊠ symbols. Thus, our invariants are maintained.
5. The total cost of performing the shift is proportional to the total size of all the zones involved $R_0, L_0, \ldots, R_{i_0}, L_{i_0}$. That is, $O(\sum_{j=0}^{i_0} 2 \cdot 2^j) = O(2^{i_0})$ operations.

After performing a shift with index i the zones $L_0, R_0, \ldots, L_{i-1}, R_{i-1}$ are half-full, which means that it will take at least $2^i - 1$ left shifts before the zones $L_0, \ldots, L_{i-1}$ become empty or at least $2^i - 1$ right shifts before the zones $R_0, \ldots, R_{i-1}$ become empty. In any case, once we perform a shift with index i, the next $2^i - 1$ shifts of that particular parallel tape will all have index less than i. This means that for every one of the parallel tapes, at most a $1/2^i$ fraction of the total number of shifts have index i. Since we perform at most T shifts, and the highest possible index is $\log T$, the total work spent in shifting $\mathcal{U}$'s k parallel tapes in the course of simulating T steps of M is

$$O(k \cdot \sum_{i=1}^{\log T} \frac{T}{2^{i-1}} 2^i) = O(T \log T) . \quad \blacksquare$$