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 [KT06, CLRS01] or Section 7.1 in Sipser's book [SIP96].

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:

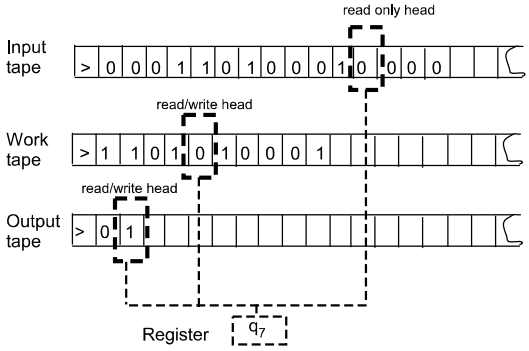

Figure 1.1: A snapshot of the execution of a 3-tape Turing machine M with an input tape, a work tape, and an output tape.

- 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 $\{0,1,\square,\triangleright\}$.

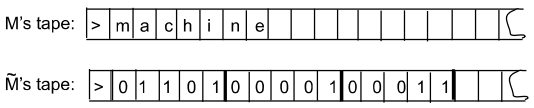

Figure 1.3: We can simulate a machine M using the alphabet $\{\triangleright,\square,\mathtt{a},\mathtt{b},\ldots,\mathtt{z}\}$ by a machine M' using $\{\triangleright,\square,0,1\}$ via encoding every tape cell of M using 5 cells of M'.

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).

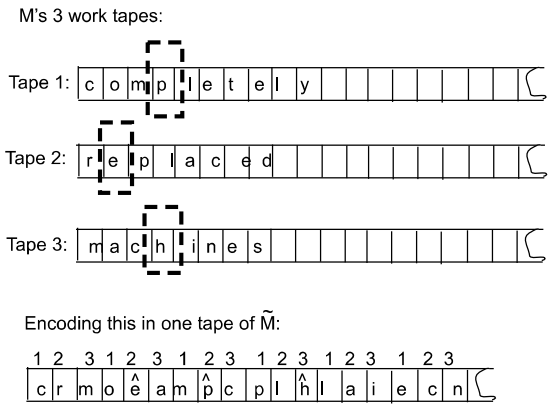

Figure 1.4: Simulating a machine M with 3 work tapes using a machine $\tilde{M}$ with a single work tape (in addition to the input and output tapes).

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 "^ 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 $\tilde{M}$.*

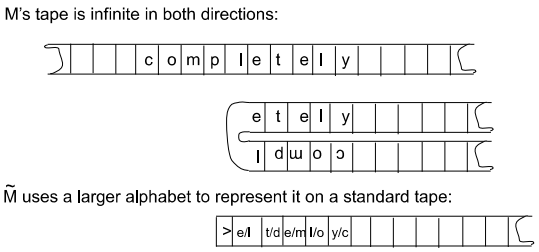

Figure 1.5: To simulate a machine M with alphabet Γ that has tapes infinite in both directions, we use a machine $\tilde{M}$ with alphabet Γ^2 whose tapes encode the "folded" version of M's tapes.

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 [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 1.5.

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 [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).

[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).

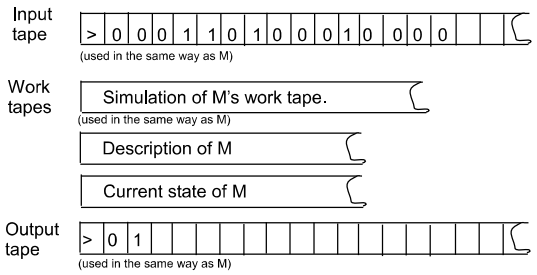

Figure 1.6: The universal TM $\mathcal{U}$ has in addition to the input and output tape, three work tapes. One work tape will have the same contents as the simulated machine M, another tape includes the description M (converted to an equivalent one-work-tape form), and another tape contains the current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

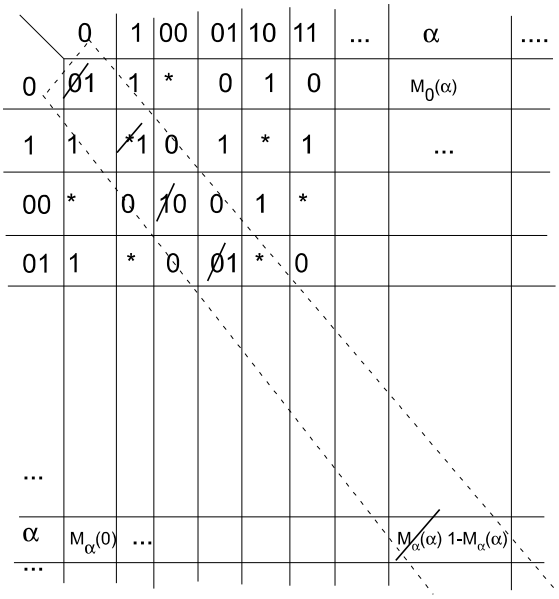

Figure 1.7: Suppose we order all strings in lexicographic order, and write in a table the value of $M_\alpha(x)$ for all strings α, x, where M_α denotes the TM represented by the string α and we use $\star$ to denote the case that $M_\alpha(x)$ is not a value in $\{0,1\}$ or that M_α does not halt on 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.