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

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

In this book we are interested in issues of *computational efficiency*, and therefore in classes of "efficiently computable" problems. Here, at first glance, it seems that we have to be very careful about our choice of a computational model, since even a kid knows that whether or not a new video game program is "efficiently computable" depends upon his computer's hardware. Surprisingly though, we can restrict attention to a single abstract computational model for studying many questions about efficiency—the Turing machine. The reason is that the Turing Machine seems able to *simulate* all

physically realizable computational models with very little loss of efficiency. Thus the set of "efficiently computable" problems is at least as large for the Turing Machine as for any other model. (One possible exception is the quantum computer model, but we do not currently know if it is physically realizable.)

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

The rest of the chapter formally defines the Turing Machine and the notion of *running time*, which is one measure of computational effort. Section 1.4 introduces a class of "efficiently computable" problems called **P** (which stands for *Polynomial* time) and discuss its philosophical significance. The section also points out how throughout the book the definition of the Turing Machine and the class **P** will be a starting point for definitions of many other models, including nondeterministic, probabilistic and quantum Turing machines, Boolean circuits, parallel computers, decision trees, and communication games. Some of these models are introduced to study arguably realizable modes of physical computation, while others are mainly used to gain insights on Turing machines.

1.1 Encodings and Languages: Some conventions

In general we study the complexity of computing a function whose input and output are finite strings of bits (i.e., members of the set $\{0,1\}^*$, see Appendix). Note that simple encodings can be used to represent general mathematical objects—integers, pairs of integers, graphs, vectors, matrices, etc.— as strings of bits. For example, we can represent an integer as a string using the binary expansion (e.g., 34 is represented as 100010) and a graph as its adjacency matrix (i.e., an n vertex graph G is represented by an $n \times n$ 0/1-valued matrix A such that $A_{i,j} = 1$ iff the edge (i,j) is present in G).

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

We will typically avoid dealing explicitly with such low level issues of representation, and will use $\llcorner x \lrcorner$ to denote some canonical (and unspecified) binary representation of the object x. Often we will drop the symbols $\llcorner \lrcorner$ and simply use x to denote both the object and its representation. We use the notation $\langle x, y \rangle$ to denote the ordered pair consisting of x and y. A canonical representation for $\langle x, y \rangle$ can be easily obtained from the representations of x and y; to reduce notational clutter, instead of $\llcorner \langle x, y \rangle \lrcorner$ we use $\langle x, y \rangle$ to denote not only the pair consisting of x and y but also the representation of this pair as a binary string.

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

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

$$\mathsf{INDSET} = \{\langle G, k \rangle : \exists S \subseteq V(G) \text{ s.t. } |S| \geq k \text{ and } \forall u, v \in S, \overline{u\,v} \notin E(G)\} \tag{1}$$

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

Big-Oh notations. As already hinted, we will often be more interested in the *rate of growth* of functions than their precise behavior. The following well known set of notations is very convenient for such analysis. 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 $g = o(f)$. For example, if $f(n) = 100n \log n$ and $g(n) = n^2$ then we have the relations $f = O(g)$, $g = \Omega(f)$, $f = \Theta(f)$, $f = o(g)$, $g = \omega(f)$. (For more examples

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

1.2 Modeling computation and efficiency

We start with an informal description of computation. Let f be a function that takes a string of bits (i.e., a member of the set $\{0,1\}^*$) and outputs, say, either 0 or 1. Informally speaking, an *algorithm* for computing f is a set of mechanical rules, such that by following them we can compute $f(x)$ given any input $x \in \{0,1\}^*$. The set of rules being followed is finite (i.e., the same set must work for all infinitely many inputs) though each rule in this set may be applied arbitrarily many times. Each rule must involve one 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.
3. Write a bit/symbol to the scratch pad.
4. Stop and output either 0 or 1.
5. Decide which of the above operations to apply based on the values that were just read.

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 machine also has

a separate tape designated as the *input* tape of the machine, whose head can only read symbols, not write them —a so-called read-only head.

The k 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 $k+1$ heads **(2)** for the k 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.

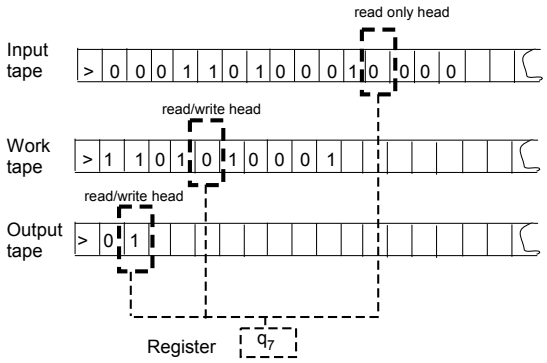

Figure 1.1: A snapshot of the execution of a 2-tape Turing machine M with an input tape, a work tape, and an output tape. We call M a 2-tape machine because the input tape can only be read from and not written to.

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

- 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 $\delta : Q \times \Gamma^{k+1} \to Q \times \Gamma^k \times \{L, R\}^{k+1}$ describing the rule M uses in performing each step. This function is called the *transition function* of M (see Figure 1.2.) (Note that δ also implicitly tells us k, the number of work tapes allowed to the TM.)

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 single-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 $(\sigma_1, \sigma_2, \ldots, \sigma_{k+1})$ are the symbols currently being read in the $k+1$ tapes, and $\delta(q, (\sigma_1, \ldots, \sigma_{k+1})) = (q', (\sigma'_2, \ldots, \sigma'_{k+1}), z)$ where $z \in \{L, R\}^{k+1}$ then at the next step the σ symbols in the last k tapes will be replaced by the σ' symbols, the machine will be in state q', and the $k+1$ heads will move left/right (i.e., L/R) 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, a finite non-blank string ("the input"), and the rest of its cells are initialized with the blank symbol. 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 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 1.1 (COMPUTING A FUNCTION AND RUNNING TIME)
Let $f : \{0,1\}^* \to \{0,1\}^*$ and let $T : \mathbb{N} \to \mathbb{N}$ be some functions. We say that a TM M *computes* function f if for every $x \in \{0,1\}^*$, if 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*[2] if for all n and all inputs x of size n, the running time of M on that input is at most $T(n)$.

Most of the specific details of our definition of Turing machines are quite arbitrary. For example, the following three 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:

CLAIM 1.2
For every $f : \{0,1\}^* \to \{0,1\}$, $T : \mathbb{N} \to \mathbb{N}$, *if* f *is computable in time* $T(n)$ *by a TM* M *using alphabet* Γ *then it is computable in time* $100 \log |\Gamma| T(n)$ *by a TM* M *using the alphabet* $\{0,1,\square,\triangleright\}$.

CLAIM 1.3
For every $f : \{0,1\}^* \to \{0,1\}$, $T : \mathbb{N} \to \mathbb{N}$, *if* f *is computable in time* $T(n)$ *by a TM* M *using* k *work tapes (plus additional input and output tapes) then it is computable in time* $100T(n)^2$ *by a TM* M *using a single work tape (plus additional input and output tapes).*

CLAIM 1.4
Define a bidirectional *TM to be a TM whose tapes are infinite in* both *directions. For every* $f : \{0,1\}^* \to \{0,1\}^*$, $T : \mathbb{N} \to \mathbb{N}$ *as above if* f *is*

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

computable in time $T(n)$ by a bidirectional TM M then it is computable in time $100T(n)$ by a standard (unidirectional) TM.

We leave the proofs of these claims as Exercises 2, 3 and 4. The reader might wish to pause at this point and work through the proofs, as this is a good way to obtain intuition for Turing machines and their capabilities.

Other changes that will not have a very significant effect include restricting the number of states to 100, having two or three dimensional tapes, allowing the machine *random access* to its tape, and making the output tape *write only* (see the texts [**?**, **?**] for proofs and more examples). In particular none of these modifications will change the class **P** of polynomial-time decision problems defined below in Section 1.4.

1.2.2 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 intuititive 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 7; also, Sipser's book [**?**] contains many more such examples.)

EXAMPLE 1.5

(This example assumes some background in computing.) We give a handwavy 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.

1.3 The Universal Turing Machine

Underlying the computer revolution of the 20th century is one simple but crucial observation: programs can be considered as strings of symbols, and hence can be given as input to other programs. The notion goes back to Turing, who described a *universal* TM that can *simulate* the execution of every other TM M given M's description as input. This enabled the construction of *general purpose* computers that are designed not to achieve one particular task, but can be loaded with a program for any arbitrary computation.

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

Now we state a computationally efficient version of Turing's construction due to Hennie and Stearns [**?**]. To give the essential idea we first prove a slightly relaxed variant where the term $t \log t$ of Condition 4 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.

THEOREM 1.6 (EFFICIENT UNIVERSAL TURING MACHINE)
There exists a TM $\mathcal{U}$ and a representation scheme of TM's satisfying:

1. *Every string $\alpha \in \{0,1\}^*$ is a representation of some TM M_α.*
2. *Every TM M is represented by infinitely many strings $\alpha \in \{0,1\}^*$.*
3. *For every $t \in \mathbb{N}$ and $x, \alpha \in \{0,1\}^*$, if on input x, the machine M_α outputs a string y within at most t steps, then $\mathcal{U}(t, x, \alpha) = y$.*
4. *On every triple $\langle t, x, \alpha \rangle$, the machine $\mathcal{U}$ runs for at most $Ct \log t$ steps, where C is a constant depending on M_α's alphabet and number of states and tapes but independent of $|\alpha|, |x|, |t|$.*

PROOF: Represent a TM M in the natural way as the tuple $\langle \gamma, q, \delta, z \rangle$ where $\gamma = |\Gamma|$ is the size of M's alphabet, q is the size of M's state space Q, the transition function δ is described by a table listing all of its inputs and outputs, and z is a table describing the elements of Γ, Q that correspond to the special symbols and states (i.e., $\triangleright, \square, 0, 1, q_{\mathsf{start}}, q_{\mathsf{halt}}$). We also allow the description to end with an arbitrary number of 1's to ensure Condition 2.[3] If a string is not a valid representation of a TM according to these rules then we consider it a representation of some canonical TM (i.e., a machine that reads its input and immediately halts and outputs 0) to ensure Condition 1.

Our universal TM $\mathcal{U}$ will use the alphabet $\{0, 1, \square, \triangleright\}$ and have, in addition to the input and output tape, five work tapes. We do not give the transition function of $\mathcal{U}$ explicitly but describe its operation in words. Suppose $\mathcal{U}$ is given the input α, t, x, where α represents some TM M. Denote by k the number of work tapes used by M. Note that if $\mathcal{U}$ were to have $k + 2$ tapes, the same alphabet as M, and more states than M, then it could trivially simulate M's execution by dedicating one tape to store the description of M and at each computational step, the universal machine can scan the transition function of M and decide how to proceed according to its rules. Thus the main difficulty is that M may use a larger number of states, a larger alphabet and more tapes than $\mathcal{U}$.

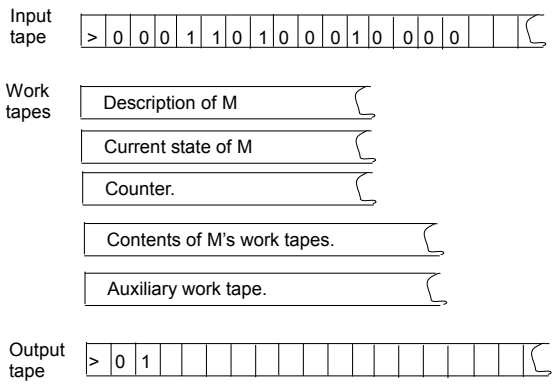

Figure 1.3: The universal TM $\mathcal{U}$ has in addition to the input and output tape, five work tapes, used to store the description of the simulated machine M, its current state, a counter that is decremented from t to 0, a tape that contains all the information in M's work tapes, and an auxiliary "scratch" work tape that is used by $\mathcal{U}$ for various computations.

[3] One can consider this convention as analogous to the *comments* feature common in many programming languages (e.g., the `/*..*/` syntax in C and Java).

These difficulties are resolved as follows (see Figure 1.3). $\mathcal{U}$ uses the input and output tapes in the same way that M uses them, and uses a single work tape —called the *main* work tape—to store the contents of all the remaining work tapes of M. Notice, each symbol of M's alphabet is represented by $\log \gamma$ bits. Furthermore, $\mathcal{U}$ uses one work tape to keep track of what state M is currently in (this only requires $\log q$ bits) and one work tape to maintain a counter (or "clock")) that counts down from t to 0. Finally, one more work tape acts as the "scratch pad" for M's own computation.

To TM $\mathcal{U}$ stores the k work tapes of M using *interleaving*: the first symbol from each of the k tapes is stored first, then the second symbol from each tape, and so on (see Figure 1.4). The symbol $\star$ marks the position of the head at each tape.

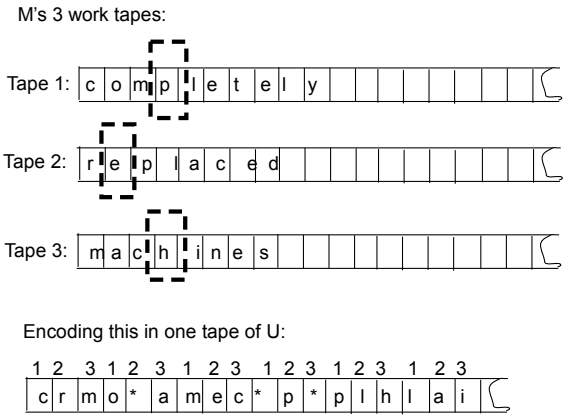

Figure 1.4: We encode k tapes into one tape by placing the contents of the first tape in positions $1, k+1, 2k+1, \ldots$, the contents of the second tape in positions $2, k+2, 2k+2, \ldots$, etc. We use the symbol $\star$ to mark the position of the head in each tape.

To simulate a single computational step of M, the machine $\mathcal{U}$ performs the following actions:

1. Scan the main work tape (the one containing the contents of M's k tapes), and copy into its scratch tape the k symbols that follow the $\star$ symbol of each tape.

2. Scan the transition function of M to find out how M behaves on these symbols (what M writes on its tapes, how it changes its state register, and how it moves the head). Then write down this information on the scratch pad.

3. Scan the main work tape and update it (both symbols written and head locations) according to the scratch pad.

4. Update the tape containing M's state according to the new state.

5. Use the same head movement and write instructions of M on the input and output tape.

6. Decrease the counter by 1, check if it has reached 0 and if so halt.

Now let's count how many computational steps $\mathcal{U}$ performs to simulate a single step of M: $\mathcal{U}$'s main tape contains at most kt symbols, and so scanning it takes $O(t)$ steps (as k is a constant depending only on M). Decreasing the counter takes $O(\log t)$ steps. The transition function, the current state, and the scratch pad only require a constant number of bits to store (where this constant depends on M's alphabet size, and number of tapes and states) and so only require a constant number of operations to read and update. Thus, simulating a single step of M takes $O(t + \log t) = O(t)$ operations, and simulating M for t steps takes $O(t^2)$ operations. ■

1.4 Deterministic time and the class P.

A *complexity class* is a set of functions that can be computed within a given resource. We will now introduce our first complexity classes. For reasons of technical convenience, throughout most of this book we will pay special attention to functions with one bit output, also known as *decision problems* or *languages*.

Definition 1.7 (The class **DTIME**.)
Let $T : \mathbb{N} \to \mathbb{N}$ be some function. We let $\mathbf{DTIME}(T(n))$ be the set of all Boolean (one bit output) functions that are computable in $c \cdot T(n)$-time for some constant $c > 0$.

Remark 1.8 (Time-constructible functions)
A function $T : \mathbb{N} \to \mathbb{N}$ is *time constructible* if the function $x \mapsto 1^{T(|x|)}$ (i.e., x is mapped to a sequence of 1's of length $T(|x|)$) is computable in $T(n)$ time. 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 typically restrict our attention to the class $\mathbf{DTIME}(T(n))$ for time-constructible T. We also typically assume that $T(n) \geq n$ as to allow the algorithm time to read its input.

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

DEFINITION 1.9 (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 INDSET has an efficient algorithm as follows: "Is INDSET $\in \mathbf{P}$?"

1.4.1 On the philosophical importance of P

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

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

We note that **P** is also a natural class from the viewpoint of a programmer. Suppose undergraduate programmers are asked to invent the definition of an "efficient" computation. Presumably, they would agree that a computation that runs in linear or quadratic time is "efficient." Next, since programmers often write programs that call other programs (or subroutines), they might find it natural to consider a program "efficient" if it performs only "efficient" computations and calls subroutines that are "efficient". The notion of "efficiency" obtained turns out to be exactly the class **P** (Cobham [**?**]). Of course, Cobham's result makes intuitive sense since composing a polynomial function with another polynomial function gives a polynomial function (for every $c, d > 0$, $(n^c)^d = n^{cd}$) but the exact proof requires some care.

1.4.2 Criticisms of P and some efforts to address them

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

Worst-case exact computation is too strict. The definition of **P** only considers algorithms that compute the function *exactly* on *every* possible input. However, not all possible inputs arise in practice (although it's not always easy to characterize the inputs that do). Chapter 15 gives a theoretical treatment of *average-case complexity* and defines the analogue of **P** in that context. Sometimes, users are willing to settle for *approximate* solutions. Chapter 19 contains a rigorous treatment of the complexity of approximation.

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

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

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

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

(Note also that we often only care about the *most significant bit* of the result, namely, a 0/1 answer.)

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

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

(c) *Use of quantum mechanics:* A more clever computational model might use some of the counterintuitive features of quantum mechanics. In Chapter 21 we define the class **BQP**, that generalizes **P** in such a way. We will see problems in **BQP** that may not be in **P**. However, currently it is unclear whether the quantum model is truly physically realizable. Even if it is realizable it currently seems only able to efficiently solve only very few "well-structured" problems that are not in **P**. Hence insights gained from studying **P** could still be applied to **BQP**.

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

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

1.4.3 Edmonds' quote

We conclude this section with a quote from Edmonds [**?**], that in the paper showing a polynomial-time algorithm for the maximum matching problem,

explained the meaning of such a result as follows:

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

WHAT HAVE WE LEARNED?

- There are many equivalent ways to mathematically model computational processes; we use the standard Turing machine formalization.
- Turing machines can be represented as strings. There is a *universal* TM that can emulate (with small overhead) any TM given its representation.
- 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.
- Most 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

The Turing Machine should be thought of as a logical construct, rather than as a piece of hardware. Most computers today are implementations of a universal computer using silicon chips. But many other physical phenomena can be used to construct universal TMs: amusing examples include bouncing billiards balls, cellular automata, and Conway's *Game of life*. It is also possible to study complexity theory axiomatically in a machine-independent fashion. See Cobham [**?**] and Blum [**?**] for two approaches.

We omitted a detailed discussion of formal complexity, and in particular the fact that the class **DTIME**$(f(n))$ can be paradoxical if f is not a *proper complexity function* (see the standard text [**?**]). We say f is proper if $f(n) \geq f(n-1)$ and there is a TM that on input x outputs a string of length $f(|x|)$ using time $O(|x| + f(|x|))$ and space $O(f(|x|))$. This notion will reappear in Chapter 4.

Exercises

§1 Prove that there exists a function $f : \{0,1\}^* \to \{0,1\}$ that is not computable in time $T(n)$ for *every* function $T : \mathbb{N} \to \mathbb{N}$.

Hint: for any string $\alpha \in \{0,1\}^*$, let M_α be the TM described by α and define f_α such that for every x, $f_\alpha(x) = 1$ if M_α on input x halts with output 1 within a finite number of steps and $f_\alpha(x) = 0$ otherwise. You need to find $f : \{0,1\}^* \to \{0,1\}$ such that for every α there exists x with $f(x) \neq f_\alpha(x)$.

§2 Prove Claim 1.2.

§3 Prove Claim 1.3.

Hint: To store the information of k tapes on a single tape, use positions $1, k+1, 2k+1, \ldots$ to store the contents of the first tape, use positions $2, k+2, 3k+2, \ldots$ to store the contents of the second tape, and so on.

§4 Prove Claim 1.4.

Hint: to simulate a bidirectional TM using alphabet size γ use a unidirectional TM of alphabet size γ^2.

§5 Define a TM M to be *oblivious* if its head movement does not depend on the input but only on the input length. That is, M is oblivious if for every input $x \in \{0,1\}^*$ and $i \in \mathbb{N}$, the location of each of M's heads at the i^{th} step of execution on input x is only a function of $|x|$ and i. Show that for every time-constructible $T : \mathbb{N} \to \mathbb{N}$, if $L \in \mathbf{DTIME}(T(n))$ then there is an oblivious TM that decides L in time $O(T(n)^2)$.

§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 $O(T(n) \log T(n))$.

Hint: show that the universal TM $\mathcal{U}$ obtained by the proof of Theorem 1.6 can be tweaked to be oblivious.

§7 Define **FDTIME** and **FP** to be the generalization of **DTIME** and **P** for non-Boolean functions (with more than one bit of output). That is, $f \in \mathbf{FDTIME}(T(n))$ if f is computable in $T(n)$ time and $\mathbf{FP} = \cup_{c \geq 1} \mathbf{FDTIME}(n^c)$.

Prove that the addition and multiplication functions are in **FP**.

Proof of Theorem 1.6: Universal Simulation in $O(t \log t)$-time

We now show how to prove Theorem 1.6 as stated, with an $O(t \log t)$ time simulation. Our machine $\mathcal{U}$ will use the same structure and number of tapes

described in Section 1.3 (see Figure 1.3). The crucial different will be the organization of the main tape of $\mathcal{U}$.

If M uses the alphabet Γ, then, as we saw before, we may assume that $\mathcal{U}$ uses the alphabet Γ^k (as this can be simulated with a constant overhead). Thus we can encode in each cell of $\mathcal{U}$'s work tape k symbols of Γ, each corresponding to a symbol from one of M's tapes. However, we still have to deal with the fact that M has k read/write heads that can each move independently to the left or right, whereas $\mathcal{U}$'s work tape only has a single head. We handle this following the dictum

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

That is, since we can not move $\mathcal{U}$'s read/write head in different directions at once, we simply move the tape "under" the head. To be more specific, since we consider $\mathcal{U}$'s work tape alphabet to be Γ^k, we can think of it as consisting of k *parallel tapes*; that is, k tapes with the property that in each step either all their read/write heads go in unison one location to the left or they all go one location to the right (see Figure 1.5). 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. For convenience, we think of $\mathcal{U}$'s parallel tapes as infinite in both the left and right directions (again, this can be easily simulated with minimal overhead, see Claim 1.4).

The approach above is still not good enough to get $O(t \log t)$-time simulation. The reason is that there may be as much as t non-blank symbols in each tape, and so each shift operation may cost $\mathcal{U}$ as much as $O(t)$ operations, resulting in $O(kt)$ operations of $\mathcal{U}$ per each step of M. Our approach to deal with this is to create "buffer zones": rather than having each of $\mathcal{U}$'s parallel tapes correspond exactly to a tape of M, we add a special kind of blank symbol $\boxtimes$ to the alphabet of $\mathcal{U}$'s parallel tapes with the semantics that this symbol is ignored in the simulation. That is, 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.

Since $\mathcal{U}$'s parallel tapes are considered bidirectional we can index their locations by $0, \pm1, \pm2, \ldots$. Normally we keep $\mathcal{U}$'s head on location 0 of these

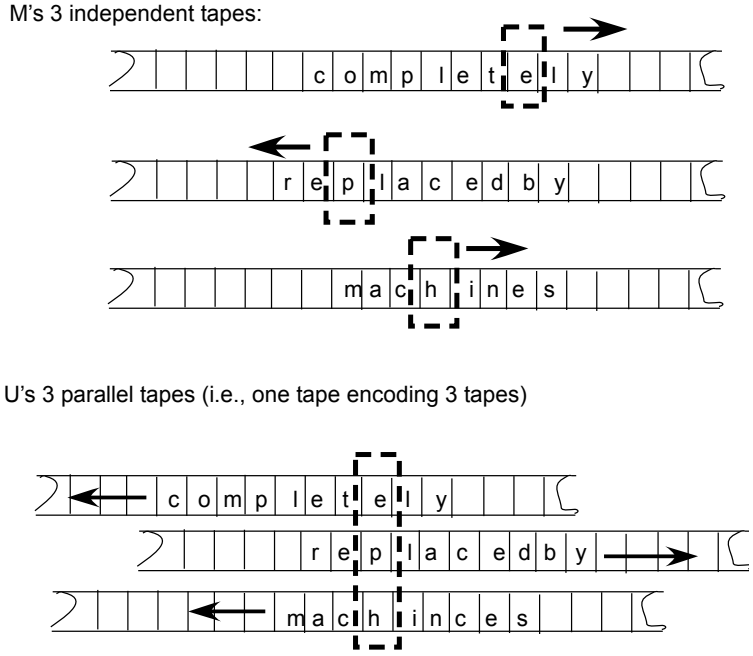

Figure 1.5: Packing k tapes of M into one tape of $\mathcal{U}$. We consider $\mathcal{U}$'s single work tape to be composed of k parallel tapes, whose heads move in unison, and hence we shift the contents of these tapes to simulate independent head movement.

parallel tapes. We will only move it temporarily to perform a shift when, following our dictum, we simulate a left head movement by shifting the tape to the right and vice versa. At the end of the shift we return the head to location 0.

We split the tapes into zones $L_1, R_1, L_2, R_2, \ldots, L_{\log t+1}, R_{\log t+1}$ where zone L_i contains the cells in the interval $[2^{i-1}+1..2^i]$ and zone R_i contains the cells in the interval $[-2^i..-2^{i-1}-1]$ (location 0 is not in any zone). Initially, we set all the zones to be half-full. That is, half of the symbols in each zones will be $\boxtimes$ and the rest will contain symbols corresponding to the work tapes of M. We always maintain the following invariants:

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

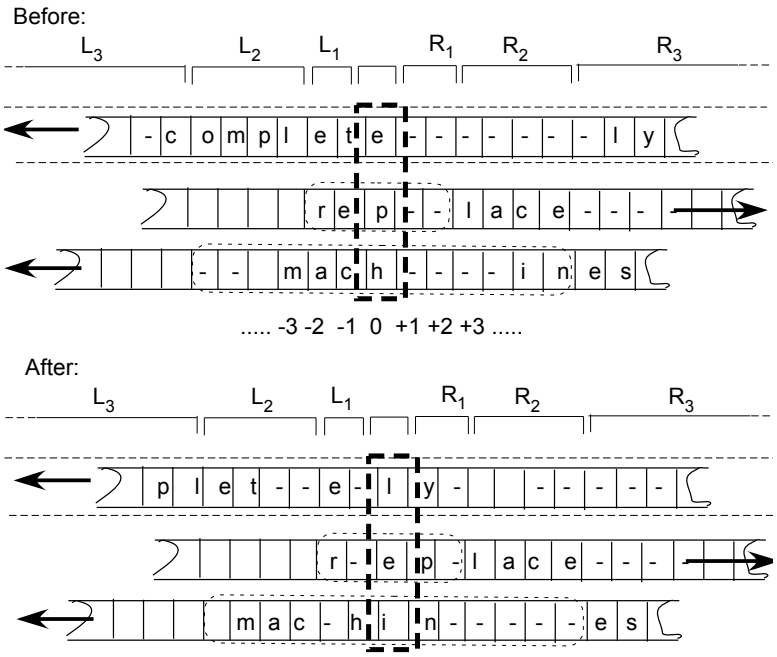

Figure 1.6: Performing a shift of the parallel tapes. The left shift of the first tape involves zones $L_1, R_1, L_2, R_2, L_3, R_3$, the right shift of the second tape involves only L_1, R_1, while the left shift of the third tape involves zones L_1, R_1, L_2, R_2. We maintain the invariant that each zone is either empty, half-full or full. Note that - denotes $\boxdot$.

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

- Location 0 always contains a non-$\boxdot$ symbol.

The advantage in setting up these zones is that now when performing the shifts, we do not always have to move the entire tape, but can restrict ourselves to only using some of the zones. We illustrate this by showing how $\mathcal{U}$ performs a left shift on the first of its parallel tapes (see Figure 1.6):

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

2. $\mathcal{U}$ puts the leftmost non-$\boxdot$ symbol of R_i in position 0 and shifts the remaining leftmost $2^{i-1} - 1$ non-$\boxdot$ symbols from R_i into the zones $R_1, \ldots, R_{i-1}$ filling up exactly half the symbols of each zone. Note

that there is room to perform this since all the zones $R_1, \ldots, R_{i-1}$ were empty and that indeed $2^{i-1} = \sum_{j=0}^{i-2} 2^j + 1$.

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

4. Note that at the end of the shift, all of the zones $L_1, R_1, \ldots, L_{i-1}, R_{i-1}$ are half-full.

Performing such a shift costs $O(\sum_{j=1}^{i} 2^j) = O(2^i)$ operations. However, once we do this, we will not touch L_i again until we perform at least 2^{i-1} shifts. Thus, we perform a shift involving L_i and R_i when simulating at most a $\frac{1}{2^{i-1}}$ of the t steps of M. We perform a shift for every one of the k tapes, but k is a constant, as is the overhead to simulate the alphabet $(\Gamma \cup \boxtimes)^k$ using the alphabet $\{0, 1, \triangleright, \square\}$ and to read the transition function and state information. Thus total number of operations used by these shifts is

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

where we need an additional $O(t \log t)$ operations to maintain the counter.[4] ■

[4] In fact, a more careful analysis shows that only $O(t)$ operations are necessary to decrease a counter from t to 0.