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 \textit{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 \textit{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\textsuperscript{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. Then we introduce a class of “efficiently computable” problems called $\textbf{P}$ (which stands for Polynomial time) and discuss its significance.

Later in the book, the definition of the Turing Machine 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.

\section{1.1 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 $\{0,1\}^\ast$) 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 $x \in \{0,1\}^\ast$. 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.

\textsuperscript{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).
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.

1.1.1 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 $\Gamma$ 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 last tape is designated as the output tape of the machine, on which it writes its final answer before halting its computation.

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.

**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, and for the $k$ read/write tapes it replaces each symbol with a new symbol (it has the option of not changing the tape by writing down the old symbol again),
2. 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
3. move each head one cell to the left or to the right.

**Formal definition.** Formally, a TM $M$ is described by a tuple containing:

- A set $\Gamma$ of the symbols that $M$’s tapes can contain. We assume that $\Gamma$ contains a designated “blank” symbol, denoted $\square$, a designated “start”
symbol, denoted $\triangleright$ and the numbers 0 and 1. We call $\Gamma$ 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 $q_{\text{start}}$ and a designated halting state, denoted $q_{\text{halt}}$.

- A function $\delta : Q \times \Gamma^{k+1} \rightarrow Q \times \Gamma^k \times \{L, R\}^{k+1}$ describing the rule $M$ uses in performing each step. This function is called the transition function of $M$.

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 $\sigma$ symbols in the last $k$ tapes will be replaced by the $\sigma'$ 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 $q_{\text{start}}$. This is called the start configuration of $M$ on input $x$.
Each step of the computation is performed by applying the function $\delta$ as
described above. The special halting state $q_{\text{halt}}$ has the property that once
the machine is in $q_{\text{halt}}$, the transition function $\delta$ does not allow it to further
modify the tape or change states. Clearly, if the machine enters $q_{\text{halt}}$ 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 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. The
running time is the number of steps in the computation.

We say that $f$ is computable in $T(n)$-time if there some machine $M$ that computes
$f$ and 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 two claims (left as Exercises 2 and 3)
show that restricting the alphabet $\Gamma$ to be $\{0,1,\square,\triangleright\}$ 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 $\Gamma$ 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**

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

---

1Formally 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. We will also use shorthand
such as $f$ is computable in $O(n^2)$-time to denote that $f$ is computable in $T(n)$ for some
function $T$ satisfying that for some constants $c,N$, $T(n) \leq c \cdot n^2$ for every $n \geq N$. 
computable in time $T(n)$ by a bidirectional TM $M$ then it is computable in time $100T(n)$ by a standard (unidirectional) TM.

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 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 1.3.

1.1.2 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 $\{0, 1\}^*$). 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 10010) and a graph as its adjacency matrix (i.e., an $n$ vertex graph $G$ is represented by an $n \times n$ 0/1-valued matrix $A$ such that $A_{i,j} = 1$ iff the edge $(i, j)$ is present in $G$).

We will typically avoid dealing explicitly with such low level issues of representation, and will use $\langle x \rangle$ to denote some canonical (and unspecified) binary representation of the object $x$. When there seems no danger of confusion, we will sometimes drop the symbols $\langle \rangle$ 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 $\langle \langle x, y \rangle \rangle$ 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:

\[ \text{INDSET} = \{ (G, k) : \exists S \subseteq V(G) \text{ s.t. } |S| \geq k \text{ and } \forall u, v \in S, u \neq v \notin E(G) \} \]

(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.

\textbf{O 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 [??].)

\subsection*{1.1.3 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 what we consider to be computation. You can get some feeling for the power of Turing machines by working 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 6; also, Sipser’s book [??] contains many more such examples.) If you have background in a particular programming language, you may try to verify that one can emulate programs in this language using a Turing machine. The following example does this for a simple tailor-made programming language:

\textbf{Example 1.4}

Consider the following simple programming language. It has a single infinite array \( A \) of elements in \( \{0, 1, \square\} \) (initialized to \( \square \)) and a single integer variable
1.2. THE UNIVERSAL TURING MACHINE

A program in this language contains a sequence of lines of the following form:

\[ \text{label : If } A[i] \text{ equals } \sigma \text{ then } \text{cmds} \]

Where \( \sigma \in \{0,1,\square\} \) and \( \text{cmds} \) is a list of one or more of the following commands: (1) Set \( A[i] \) to \( \tau \) where \( \tau \in \{0,1,\square\} \), (2) Goto label, (3) Increment \( i \) by one, (4) Decrement \( i \) by one, and (5) Halt.

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. It is not hard to emulate such a program by a Turing machine using the following transformation:

<table>
<thead>
<tr>
<th>Program</th>
<th>Turing machine</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program counter / current line</td>
<td>State register of TM</td>
</tr>
<tr>
<td>Memory / array</td>
<td>Tapes of TM</td>
</tr>
</tbody>
</table>

Similar ideas are used to emulate the execution of common programming languages such as C, Java, etc., using Turing machines.

1.2 The Universal Turing Machine

Underlying the computer revolution of the 20th century is one crucial observation: programs can be considered also as lists of symbols, or strings, and hence given as input to other programs. This enabled the construction of general purpose computers that are not designed in advance to achieve one particular task, but rather can be loaded with a program for any arbitrary computation. This observation was first made by Turing in the context of his machines. He showed that every TM can be represented as a string, and that there exists a single universal TM that can emulate the execution of every TM \( M \) given \( M \)'s description as input. We will use this observation in several places in this book. Below we give a computationally efficient version of Turing’s Theorem that was proven by Hennie and Stearns \([?]\).
1.2. THE UNIVERSAL TURING MACHINE

**Theorem 1.5 (Universal Turing machine)**

There exists a TM $U$ and a representation scheme of TM’s that satisfy the following conditions:

1. Every string $\alpha \in \{0,1\}^*$ is a representation of some TM $M_\alpha$.

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 $M_\alpha$ outputs a string $y$ within at most $t$ steps, then on inputs $\langle t, x, \alpha \rangle$ the machine $U$ outputs $y$.

4. On every triple $\langle t, x, \alpha \rangle$, the machine $U$ runs for at most $Ct \log t$ steps, where $C$ is a constant depending on $M_\alpha$’s alphabet and number of states and tapes but independent of $|\alpha|, |x|, |t|$.

**Remark 1.6**

The proof below goes to some effort to achieve the best known bound on the running time of $U$. It is significantly easier to prove Theorem 1.5 if one replaces the term $t \log t$ of Condition 4 with, say, $t^2$ or $t^3$, and such a bound (of the form $t^c$ for some constant $c$) suffices for most applications of this theorem. The reader may wish to try proving such a result as an exercise.

**Proof:** We represent a TM $M$ in the natural way as the tuple $\langle \gamma, q, \delta, z \rangle$ where $\gamma$ is the size of $M$’s alphabet $\Gamma$, $q$ is the size of $M$’s state space $Q$, the transition function $\delta$ is described by a table listing all of its inputs and outputs, and $z$ is a table describing the elements of $\Gamma, Q$ that correspond to the special symbols and states (i.e., $\triangleright, \square, 0, 1, q_{\text{start}}, q_{\text{halt}}$). We also allow the description to end with an arbitrary number of 1’s to ensure Condition 2.

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. Note that the length of the description of a machine with $k$ tapes, alphabet size $\gamma$ and $q$ states is at least $|Q||\Gamma|^k$.

Our universal TM $U$ will have, in addition to the input and output tape, four work tapes and alphabet $\{0,1,\square,\triangleright\}$. Given an input triple $\alpha, t, x$ we denote by $M$ the TM represented by $\alpha$. We denote by $k$ the number of

---

\[2\] One can consider this convention as analogous to the comments feature common in many programming languages (e.g., the /*...*/ syntax in C and Java).
tapes that $M$ uses aside from the input and output tapes. It is not hard to simulate $M$’s execution using, say, $k + 2$ tapes, the same alphabet as $M$, and more states than $M$: at each computational step, the universal machine can scan the transition function of $M$ and decide how to proceed according to its rules. We actually do not even need more states than $M$: we can store use one of the tapes to store the current state and so have a universal machine $U$ with a constant number of states (independent of the states of the machine simulated). The main challenges in the simulation are that $M$ may use both a larger alphabet and more tapes than $U$.

Simulating a machine with larger alphabet is not an issue: as shown in Claim 1.2 this can easily done with $O(\log |\Gamma|) = \text{constant overhead}$. The tricky part is to simulate a machine that uses more tapes. Even this is not complicated if we were willing to replace the quantity $t \log t$ with $t^2$ in Condition 4. To simulate $k$ tapes on one tape, we could place the contents of the first tape (including some special symbol to depict the position of the head) on positions $1, k + 1, 2k + 1, \ldots$, the contents of the second tape on positions $2, k + 2, 2k + 2, \ldots$, etc. The problem is that to simulate one step of $M$, the machine $U$ will need to make a pass over all of its tape, which could take as much as $kt$ operations, leading to $kt^2 = O(t^2)$ time simulation.

To simulate in $t \log t$ time, we will have to be more careful. One tape of $U$ will be used to maintain a counter counting from $t$ backwards. Another tape will be used to keep $M$ and the current state that $M$ is in. A third tape will be the main work tape containing all the information of $M$’s $k$ work tapes. A fourth tape will be an auxiliary tape that will be helpful in copying parts of the work tape from one place to another. The machine $U$ will place its input head on the start of the encoding of $x$ and will use the input and output tapes in the same way $M$ does.

How can we pack all of $M$’s $k$ work tapes into a single tape of $U$? For starters, as we saw above, we can assume that $U$ uses in this tape the alphabet $\Gamma^k$ without adding too much of an overhead. Thus we can encode in each cell of $U$’s work tape $k$ symbols of $\Gamma$, 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 $U$’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 $U$’s read/write head in different directions at once, we simply move the tape “under” the head. To be more specific, since we consider $U$’s work tape alphabet to be $\Gamma^k$, 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 1.2). 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 $U$ 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 $U$’s parallel tapes as infinite in both the left and right directions (again, this can be easily simulated with minimal overhead, see Claim 1.3).

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 $U$ as much as $O(t)$ operations, resulting in $O(kt)$ operations of $U$ per each step of $M$. Our approach to deal with this will be to create “buffer zones”: rather than having each of
1.2. THE UNIVERSAL TURING MACHINE

\[ \text{Before:} \]
\[ \begin{array}{cccccccc}
\text{L}_3 & \text{L}_2 & \text{L}_1 & \text{R}_1 & \text{R}_2 & \text{R}_3 \\
\end{array} \]

\[ \vdots \]

\[ \begin{array}{cccccccc}
\text{R} & \text{L} & \text{R} & \text{L} & \text{R} & \text{L}
\end{array} \]

\[ \vdots \]

\[ \text{After:} \]
\[ \begin{array}{cccccccc}
\text{L}_3 & \text{L}_2 & \text{L}_1 & \text{R}_1 & \text{R}_2 & \text{R}_3 \\
\end{array} \]

\[ \vdots \]

\[ \begin{array}{cccccccc}
\text{R} & \text{L} & \text{R} & \text{L} & \text{R} & \text{L}
\end{array} \]

\[ \vdots \]

\[ \begin{array}{cccccccc}
\text{L} & \text{R} & \text{L} & \text{R} & \text{L} & \text{R}
\end{array} \]

\[ \vdots \]

Figure 1.3: 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 □.

\( U \)'s parallel tapes correspond exactly to a tape of \( M \), we add a special kind of blank symbol □ to the alphabet of \( U \)'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 \( U \) not just by 010 but also by 0□1 or 0□□1□0 and so on.

Since \( U \)'s parallel tapes are considered bidirectional we can index their locations by \( 0, \pm 1, \pm 2, \ldots \). Normally we keep \( U \)'s head on location 0 of these 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
1.2. THE UNIVERSAL TURING MACHINE

each zones will be \( \square \) 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 \( \square \) 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 \( \Gamma \) and in particular a zone full of \( \square \)'s is considered
  full.)

- The total number of non-\( \square \) 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-\( \square \) 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
\( U \) performs a left shift on the first of its parallel tapes (see Figure 1.3):

1. \( U \) 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. \( U \) puts the leftmost non-\( \square \) symbol of \( R_i \) in position 0 and shifts the
   remaining leftmost \( 2^{i-1} - 1 \) non-\( \square \) 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. \( U \) 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 reorga-
   nizes \( L_{i-1}, \ldots, L_1 \) 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
1.3. 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 $\text{DTIME}(T(n))$ 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 $\text{DTIME}(T(n))$ 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)**

$P = \bigcup_{c \geq 1} \text{DTIME}(n^c)$

Thus, we can phrase the question from the introduction as to whether $\text{INDSET}$ has an efficient algorithm as follows: “Is $\text{INDSET} \in P$?”
1.3. Deterministic Time and the Class \( \mathbf{P} \).

1.3.1 On the philosophical importance of \( \mathbf{P} \).

The class \( \mathbf{P} \) is by no means a perfect formalization of the notion of efficient algorithms, but it will serve us quite well in our study. \( \mathbf{P} \) has been criticized as being both too permissive (allowing algorithms that are obviously inefficient) and too restrictive (not allowing efficient modes of computation in the real life.) We address now some of these criticisms.

Polynomials can be too large. One criticism of \( \mathbf{P} \) is that it allows functions whose best algorithm takes, say \( n^{100} \) or \( n^2 + 10^{20} \) time to compute. It seems quite a stretch to call such functions efficiently computable. Nevertheless, learning that a function \( f \) has an \( n^{100} \)-time algorithm as opposed to a \( 2^n \)-time algorithm is still an important insight on \( f \). Indeed, often in the past such insights led eventually to more efficient (e.g., \( n^2 \) or \( n^3 \) with reasonable constants) algorithms. In any case, \( \mathbf{P} \) is not intended to be synonymous with efficient computation, but rather to separate the grossly inefficient algorithms from the truly efficient.

One justification for the particular choice of polynomials is that they are closed under composition (i.e., if \( T(n) \leq n^c \) and \( T'(n) \leq n^d \) then \( T'(T(n)) \leq n^{cd} \)) which allows for certain robustness in \( \mathbf{P} \)'s definition.

Regardless, a definition is never more than a tool. When studying algorithms for a particular problem the goal strongly depends on the context and our current knowledge: an \( n^5 \) algorithm is often not good enough as a solution to a practical problem but an \( n^{\log^{10} n} \)-time algorithm for \textsc{Indset} would be a fantastic breakthrough.

\( \mathbf{P} \) does not capture important practical considerations. In reality, the efficiency of an algorithm is never just a number, and depends on many properties of the execution environment. Not all of these properties are equally amenable to theoretical analysis. However, as we will see in Chapters 3 and 6 there are complexity classes such as \( \mathbf{L} \) and \( \mathbf{NC} \) that are more restrictive than \( \mathbf{P} \) and aim to capture important considerations such as memory usage and parallelism.

Worst-case exact computation is too strict. The definition of \( \mathbf{P} \) 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. Sometimes, users are willing to settle for approximate solutions. Chapter 19 contains a rigorous treatment of the complexity of approximation.
1.3. DETERMINISTIC TIME AND THE CLASS P.

Other physically realizable models. The Church-Turing (CT) thesis 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. The CT thesis is definitely not a mathematical statement, and indeed it is not even a precise statement about our finite physical world, but it still seems to be “morally true” in the sense that is compatible with our current understanding, based on computer science, logic and the natural sciences, of all known and conceivable devices of computation.

The strong form of the CT thesis is that every physically realizable computation device can be simulated by a TM with polynomial overhead. In particular it implies that such devices can not compute in feasible time a function outside of \( \mathbf{P} \). Again, this is neither a mathematical or physical statement, although, assuming we will eventually be able to prove some strong circuit lower bounds (see Chapters 6 and 13), we may be able to derive from it some refutable physical hypothesis.

The strong CT thesis is more controversial, and there have been some suggestions for possibly realizable computational models that may compute functions outside of \( \mathbf{P} \). One such model is a probabilistic algorithm that can toss coins. The complexity class \( \mathbf{BPP} \) (see Chapter 7) consists of the functions computable in polynomial time by such algorithms. However, we will see in Chapter 17 that it is likely that \( \mathbf{BPP} = \mathbf{P} \). Another, perhaps more serious attack on the strong CT thesis, is the notion of quantum computers: devices that utilize some of the paradoxical-sounding notions of quantum mechanics such as interference to achieve more computational power. The complexity class \( \mathbf{BQP} \) (see Chapter 21) consists of the functions computable in polynomial time by such algorithms. We do not know whether \( \mathbf{BQP} = \mathbf{P} \) and there have been candidates for functions in \( \mathbf{BQP} \setminus \mathbf{P} \) (i.e., integer factorization). However, it still not clear whether this model is truly physically realizable. Even if it is realizable and it does hold that \( \mathbf{BQP} \neq \mathbf{P} \), our current intuition is that \( \mathbf{BQP} \) is “close” to \( \mathbf{P} \) in the sense that, aside from few structured problems, their computational powers are not that different. Hence insights gained from studying \( \mathbf{P} \) could be applied to \( \mathbf{BQP} \).

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.

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 \( \mathbf{P} \) consists of all decision problems that are solvable by Turing machines in polynomial time. We say that problems in \( \mathbf{P} \) 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 \( \mathbf{P} \).

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 \( \text{DTIME}(f(n)) \) 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

\[ §1 \text{ Prove that there exists a function } f : \{0,1\}^* \rightarrow \{0,1\} \text{ that is not computable in time } T(n) \text{ for every function } T : \mathbb{N} \rightarrow \mathbb{N}. \]
1.3. DETERMINISTIC TIME AND THE CLASS P.

§2 Prove Claim 1.2.

§3 Prove Claim 1.3.

§4 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 $T : \mathbb{N} \rightarrow \mathbb{N}$, if $L \in \text{DTIME}(T(n))$ then there is an oblivious TM that decides $L$ in time $O(T(n)^2)$.

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

§6 Define $\text{FDTIME}$ and $\text{FP}$ to be the generalization of $\text{DTIME}$ and $\text{P}$ for non-Boolean functions (with more than one bit of output). That is, $f \in \text{FDTIME}(T(n))$ if $f$ is computable in $T(n)$ time and $\text{FP} = \bigcup_{c \geq 1} \text{FDTIME}(n^c)$. Prove that the addition and multiplication functions are in $\text{FP}$. 

\[ (x) \neq (x) \text{ for all } x \]
1.3. DETERMINISTIC TIME AND THE CLASS P.