CHAPTER 0 Notational conventions

We now specify some of the notations and conventions used throughout this book. We make use of some notions from discrete mathematics such as strings, sets, functions, tuples, and graphs. All of these are reviewed in Appendix A.

Standard notation

We let $\mathbb{Z} = \{0, \pm 1, \pm 2, \dots\}$ denote the set of integers, and $\mathbb{N}$ denote the set of natural numbers (i.e., nonnegative integers). A number denoted by one of the letters i, j, k, ℓ, m, n is always assumed to be an integer. If $n \geq 1$, then $[n]$ denotes the set $\{1, \dots, n\}$. For a real number x, we denote by $\lceil x \rceil$ the smallest $n \in \mathbb{Z}$ such that $n \geq x$ and by $\lfloor x \rfloor$ the largest $n \in \mathbb{Z}$ such that $n \leq x$. Whenever we use a real number in a context requiring an integer, the operator $\lceil\ \rceil$ is implied. We denote by $\log x$ the logarithm of x to the base 2. We say that a condition $P(n)$ holds for *sufficiently large* n if there exists some number N such that $P(n)$ holds for every $n > N$ (for example, $2^n > 100n^2$ for sufficiently large n). We use expressions such as $\sum_i f(i)$ (as opposed to, say, $\sum_{i=1}^{n} f(i)$) when the range of values i takes is obvious from the context. If u is a string or vector, then u_i denotes the value of the i^{th} symbol/coordinate of u.

Strings

If S is a finite set then a *string* over the alphabet S is a finite ordered tuple of elements from S. In this book we will typically consider strings over the *binary* alphabet $\{0, 1\}$. For any integer $n \geq 0$, we denote by S^n the set of length-n strings over S (S^0 denotes the singleton consisting of the empty tuple). We denote by S^* the set of all strings (i.e., $S^* = \cup_{n \geq 0} S^n$). If x and y are strings, then we denote their concatenation (the tuple that contains first the elements of x and then the elements of y) by $x \circ y$ or sometimes simply xy. If x is a string and $k \geq 1$ is a natural number, then x^k denotes the concatenation of k copies of x. For example, 1^k denotes the string consisting of k ones. The length of a string x is denoted by $|x|$.

Additional notation

If S is a distribution then we use $x \in_{\text{R}} S$ to say that x is a random variable that is distributed according to S; if S is a set then this denotes that x is distributed uniformly over the members of S. We denote by U_n the uniform distribution over $\{0, 1\}^n$. For two length-n strings $x, y \in \{0, 1\}^n$, we denote by $x \odot y$ their dot product modulo 2; that is $x \odot y = \sum_i x_i y_i \pmod 2$. In contrast, the inner product of two n-dimensional real or complex vectors $\mathbf{u}, \mathbf{v}$ is denoted by $\langle \mathbf{u}, \mathbf{v} \rangle$ (see Section A.5.1). For any object x, we use $\llcorner x \lrcorner$ (not to be confused with the floor operator $\lfloor x \rfloor$) to denote the representation of x as a string (see Section 0.1).

0.1 REPRESENTING OBJECTS AS STRINGS

The basic computational task considered in this book is *computing a function*. In fact, we will typically restrict ourselves to functions whose inputs and outputs are finite *strings of bits* (i.e., members of $\{0, 1\}^*$).

Representation

Considering only functions that operate on bit strings is not a real restriction since simple encodings can be used to *represent* general objects—integers, pairs of integers, graphs, vectors, matrices, etc.—as strings of bits. For example, we can represent an integer as a string using the binary expansion (e.g., 34 is represented as 100010) and a graph as its adjacency matrix (i.e., an n vertex graph G is represented by an $n \times n$ 0/1-valued matrix A such that $A_{i,j} = 1$ iff the edge $\overline{i\,j}$ is present in G). We will typically avoid dealing explicitly with such low-level issues of representation and will use $\llcorner x \lrcorner$ to denote some canonical (and unspecified) binary representation of the object x. Often we will drop the symbols $\llcorner\,\lrcorner$ and simply use x to denote both the object and its representation.

Representing pairs and tuples

We use the notation $\langle x, y \rangle$ to denote the ordered pair consisting of x and y. A canonical representation for $\langle x, y \rangle$ can be easily obtained from the representations of x and y. For example, we can first encode $\langle x, y \rangle$ as the string $\llcorner x \lrcorner \# \llcorner y \lrcorner$ over the alphabet $\{0, 1, \#\}$ and then use the mapping $0 \mapsto 00, 1 \mapsto 11, \# \mapsto 01$ to convert this representation into a string of bits. To reduce notational clutter, instead of $\llcorner \langle x, y \rangle \lrcorner$ we use $\langle x, y \rangle$ to denote not only the pair consisting of x and y but also the representation of this pair as a binary string. Similarly, we use $\langle x, y, z \rangle$ to denote both the ordered triple consisting of x, y, z and its representation, and use similar notation for 4-tuples, 5-tuples, etc.

Computing functions with nonstring inputs or outputs

The idea of representation allows us to talk about computing functions whose inputs are not strings (e.g., functions that take natural numbers as inputs). In all these cases, we implicitly identify any function f whose domain and range are not strings with the function $g : \{0, 1\}^* \to \{0, 1\}^*$ that given a representation of an object x as input, outputs the representation of $f(x)$. Also, using the representation of pairs and tuples, we can also talk about computing functions that have more than one input or output.

0.2 DECISION PROBLEMS/LANGUAGES

An important special case of functions mapping strings to strings is the case of *Boolean* functions, whose output is a single bit. We identify such a function f with the subset $L_f = \{x : f(x) = 1\}$ of $\{0, 1\}^*$ and call such sets *languages* or *decision problems* (we use these terms interchangeably).[1] We identify the computational problem of computing f (i.e., given x compute $f(x)$) with the problem of deciding the language L_f (i.e., given x, decide whether $x \in L_f$).

EXAMPLE 0.1

By representing the possible invitees to a dinner party with the vertices of a graph having an edge between any two people who don't get along, the dinner party computational problem from the introduction becomes the problem of finding a maximum sized *independent set* (set of vertices without any common edges) in a given graph. The corresponding language is:

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

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

0.3 BIG-OH NOTATION

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

Definition 0.2 *(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

[1] The word "language" is perhaps not an ideal choice to denote subsets of $\{0, 1\}^*$, but for historical reasons this is by now standard terminology.

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 0.3

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 $f(n) = 100n^2 + 24n + 2\log n$ 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, $h(n) \leq n^c$. We'll sometimes also also write $h(n) = \text{poly}(n)$ in this case.

For more examples and explanations, see any undergraduate algorithms text such as [DPV06, KT06, CLRS01] or Section 7.1 in Sipser's book [Sip96].

EXERCISES

0.1. For each of the following pairs of functions f, g determine whether $f = o(g)$, $g = o(f)$ or $f = \Theta(g)$. If $f = o(g)$ then find the first number n such that $f(n) < g(n)$:
(a) $f(n) = n^2$, $g(n) = 2n^2 + 100\sqrt{n}$.
(b) $f(n) = n^{100}$, $g(n) = 2^{n/100}$.
(c) $f(n) = n^{100}$, $g(n) = 2^{n^{1/100}}$.
(d) $f(n) = \sqrt{n}$, $g(n) = 2^{\sqrt{\log n}}$.
(e) $f(n) = n^{100}$, $g(n) = 2^{(\log n)^2}$.
(f) $f(n) = 1000n$, $g(n) = n \log n$.

0.2. For each of the following recursively defined functions f, find a closed (nonrecursive) expression for a function g such that $f(n) = \Theta(g(n))$, and prove that this is the case. (*Note*: Below we only supply the recursive rule, you can assume that $f(1) = f(2) = \cdots = f(10) = 1$ and the recursive rule is applied for $n > 10$; in any case, regardless of how the base case is defined it won't make any difference to the answer. Can you see why?)
(a) $f(n) = f(n-1) + 10$.
(b) $f(n) = f(n-1) + n$.
(c) $f(n) = 2f(n-1)$.

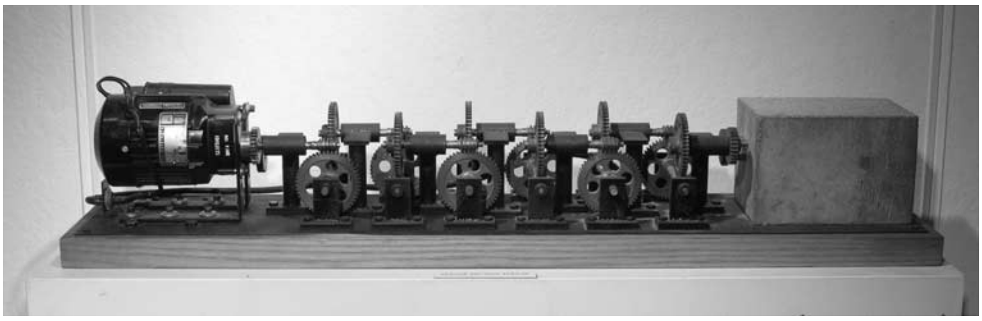

Figure 0.1. *Machine with Concrete* by Arthur Ganson. Reproduced with permission of the artist.

(d) $f(n) = f(n/2) + 10$.
(e) $f(n) = f(n/2) + n$.
(f) $f(n) = 2f(n/2) + n$.
(g) $f(n) = 3f(n/2)$.
(h) $f(n) = 2f(n/2) + O(n^2)$.

H531

0.3. The MIT museum contains a kinetic sculpture by Arthur Ganson called *Machine with Concrete* (see Figure 0.1). It consists of 13 gears connected to one another in a series such that each gear moves 50 times slower than the previous one. The fastest gear is constantly rotated by an engine at a rate of 212 rotations per minute. The slowest gear is fixed to a block of concrete and so apparently cannot move at all. Explain why this machine does not break apart.

PART ONE BASIC COMPLEXITY CLASSES

CHAPTER 1 The computational model—and why it doesn't matter

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

– Alan Turing, 1950

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

– Kurt Gödel, 1946

The problem of mathematically modeling computation may at first seem insurmountable: Throughout history people have been solving computational tasks using a wide variety of methods, ranging from intuition and "eureka" moments to mechanical devices such as abacus or sliderules to modern computers. Besides that, other organisms and systems in nature are also faced with and solve computational tasks every day using a bewildering array of mechanisms. How can you find a simple mathematical model that captures all of these ways to compute? The problem is even further exacerbated since in this book we are interested in issues of *computational efficiency*. Here, at first glance, it seems that we have to be very careful about our choice of a computational model, since even a kid knows that whether or not a new video game program is "efficiently computable" depends upon his computer's hardware.

Surprisingly enough, it turns out there there is a simple mathematical model that suffices for studying many questions about computation and its efficiency—the *Turing machine*. It suffices to restrict attention to this single model since it seems able to *simulate* all physically realizable computational methods with little loss of efficiency. Thus the set of "efficiently solvable" computational tasks is at least as large for the Turing machine as for any other method of computation. (One possible exception is the quantum computer model described in Chapter 10, but we do not currently know if it is physically realizable.)

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

Since complexity theory is concerned with *computational efficiency*, Section 1.6 contains one of the most important definitions in this book: the definition of complexity class **P**, which aims to capture mathematically the set of all decision problems that can be efficiently solved. Section 1.6 also contains some discussion on whether or not the class **P** truly captures the informal notion of "efficient computation." The section also points out how throughout the book the definition of the Turing machine and the class **P** will be a starting point for definitions of many other models, including nondeterministic, probabilistic, and quantum Turing machines, Boolean circuits, parallel computers, decision trees, and communication games. Some of these models are introduced to study arguably realizable modes of physical computation, while others are mainly used to gain insights on Turing machine computations.

1.1 MODELING COMPUTATION: WHAT YOU REALLY NEED TO KNOW

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

For thousands of years, the term "computation" was understood to mean application of mechanical rules to manipulate numbers, where the person/machine doing the manipulation is allowed a *scratch pad* on which to write the intermediate results. The Turing machine is a concrete embodiment of this intuitive notion. Section 1.2.1 shows that it can be also viewed as the equivalent of any modern programming language—albeit one with no built-in prohibition on its memory size.[1]

Here we describe this model informally along the lines of Turing's quote at the start of the chapter. Let f be a function that takes a string of bits (i.e., a member of the set $\{0, 1\}^*$) and outputs either 0 or 1. An *algorithm* for computing f is a set of mechanical rules, such that by following them we can compute $f(x)$ given any input $x \in \{0, 1\}^*$. The set of rules being followed is fixed (i.e., the same rules must work for all possible inputs) though each rule in this set may be applied arbitrarily many times. Each rule involves one or more of the following "elementary" operations:

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

Based on the values read,

1. Write a bit/symbol to the scratch pad.
2. Either stop and output 0 or 1, or choose a new rule from the set that will be applied next.

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

Finally, the *running time* is the number of these basic operations performed. We measure it in asymptotic terms, so we say a machine runs in time $T(n)$ if it performs at most $T(n)$ basic operations time on inputs of length n.

The following are simple facts about this model.

1. The model is robust to almost any tweak in the definition such as changing the alphabet from $\{0,1,\ldots,9\}$ to $\{0,1\}$, or allowing multiple scratchpads, and so on. The most basic version of the model can *simulate* the most complicated version with at most polynomial (actually quadratic) slowdown. Thus t steps on the complicated model can be simulated in $O(t^c)$ steps on the weaker model where c is a constant depending only on the two models. See Section 1.3.
2. An algorithm (i.e., a machine) can be represented as a bit string once we decide on some canonical encoding. Thus an algorithm/machine can be viewed as a possible *input* to another algorithm—this makes the boundary between *input*, *software*, and *hardware* very fluid. (As an aside, we note that this fluidity is the basis of a lot of computer technology.) We denote by M_α the machine whose representation as a bit string is α.
3. There is a *universal* Turing machine $\mathcal{U}$ that can *simulate* any other Turing machine given its bit representation. Given a pair of bit strings (x,α) as input, this machine simulates the behavior of M_α on input x. This simulation is very efficient: If the running time of M_α was $T(|x|)$, then the running time of $\mathcal{U}$ is $O(T(|x|)\log T(|x|))$. See Section 1.4.
4. The previous two facts can be used to easily prove the existence of functions that are not computable by any Turing machine; see Section 1.5. Uncomputability has an intimate connection to Gödel's famous Incompleteness Theorem; see Section 1.5.2.

1.2 THE TURING MACHINE

The *k-tape Turing machine* (TM) concretely realizes the above informal notion in the following way (see Figure 1.1).

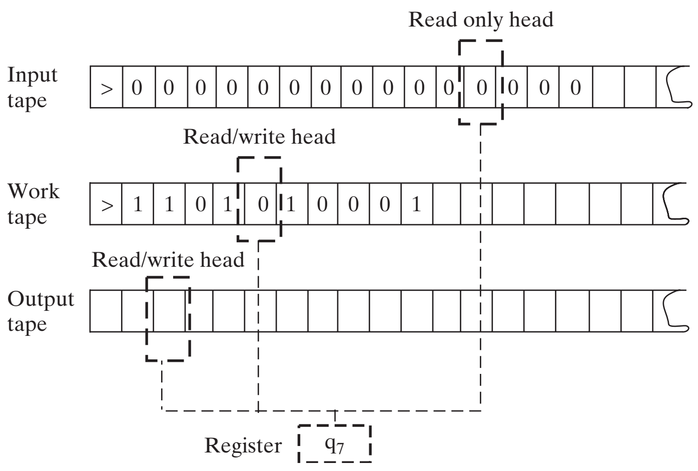

Figure 1.1. A snapshot of the execution of a three-tape Turing machine M with an input tape, a work tape, and an output tape.

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

The first tape of the machine is designated as the *input* tape. The machine's head can only read symbols from that tape, not write them—a so-called read-only head. The $k - 1$ read-write tapes are called *work tapes*, and the last one of them is designated as the *output tape* of the machine, on which it writes its final answer before halting its computation.

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

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

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

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

- A finite set Γ of the symbols that M's tapes can contain. We assume that Γ contains a designated "blank" symbol, denoted $\square$; a designated "start" symbol, denoted $\triangleright$; and the numbers 0 and 1. We call Γ the *alphabet* of M.
- A finite set Q of possible states M's register can be in. We assume that Q contains a designated start state, denoted $q_{\mathtt{start}}$, and a designated halting state, denoted $q_{\mathtt{halt}}$.

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