Basic Notions in Computational Complexity

Chapter 24 of the forthcoming *CRC Handbook on Algorithms and Theory of Computation*

Tao Jiang
Department of Computer Science
McMaster University
Hamilton, Ontario L8S 4K1, Canada

Ming Li
Department of Computer Science
University of Waterloo
Waterloo, Ontario N2L 3G1, Canada

Bala Ravikumar
Department of Computer Science
University of Rhode Island
Kingston, RI 02881, USA

1 Introduction

Computational complexity is aimed at measuring how complex a computational solution is. There are many ways to measure the complexity of a solution: how hard it is to understand it, how hard to express it, how long the solution process will take, and more. The last criterion—time—is most widely taken as the definition of complexity. This is because the agent that implements an algorithm is usually a computer—and from a user's point of view, the most important issue is how long one should wait to see the solution. However, there are other important measures of complexity, such as the amount of memory, hardware, information, communication, knowledge, or energy needed for the solution. Complexity theory is aimed at quantifying these resources precisely and studying the amounts of them required to accomplish computational tasks.

This technical sense of the word "complexity" did not take root until the mid-1960s. Today, complexity theory is not only a vibrant subfield of computer science, but also has direct or metaphoric impact on other fields such as dynamical systems and chaos theory.

The seed of computational complexity is the formalization of the concept of an **algorithm**. Algorithms in turn must be planted in some *computational model*, ideally one that abstracts important features of real computing machines and processes. In this chapter we consider the **Turing machine**, a computer model created and studied by the British mathematician Alan Turing in the 1930s [Turing, 1936]. Chapter 26 will discuss different (and equivalent) ways of formalizing algorithms.

With the advent of commercial computers in the 1950s and 1960s, when processor speed was much lower and memory cost unimaginably higher than today, it became critical to design *efficient* algorithms for solving large classes of problems. Just knowing that a problem was solvable, which had been the main concern of computability theory since the 1930s, was no longer enough. Turing machines provided a basis for Hartmanis and Stearns [Hartmanis and Stearns, 1965] to formally define the measures of **time complexity** and **space complexity** for computations. The latter refers to the amount of memory needed to execute the computation. They defined measures for other resources as well, and Blum [Blum, 1967] found a precise definition of *complexity measure* that was not tied to any particular resource or machine model. Together with earlier work by Rabin [Rabin, 1963], these papers marked the beginning of *computational complexity theory* as an important new discipline. A closely related development was drawn together by Knuth, whose work on algorithms and data structures [Knuth, 1969] created the field of *algorithm design and analysis*. All of this work has been recognized in annual Turing Awards given by the Association for Computing Machinery. Hartmanis' Turing Award lecture [Hartmanis, 1994] recounts the origins of computational complexity theory and speculates on its future development.

The power of a computing machine depends on its internal structure as well as on the amount of time, space, or other resources it is allowed to consume. Restricting our model of choice, the Turing machine, in various internal ways yields progressively simpler and weaker computing machines. These machines correspond to a natural hierarchy of grammars defined and advanced by Chomsky [Chomsky, 1956], which we describe in Chapter 25. In this chapter we present the most basic computational models, and use these models to classify problems first by solvability and then by complexity.

Central issues studied by researchers in computational complexity include the following.

- For a given amount of resources, or for a given type of resource, what problems can and cannot be solved?
- What is the relationship between problems requiring essentially the same amount of the resource or resources? May they be equivalent in some intrinsic sense?
- What connections are there between different kinds of resources? Given more time, can we reduce the demand on memory storage, or vice-versa?
- What general limits can be set on the kind of problems that can be solved when resources are limited?

The end thrust of Turing's famous paper [Turing, 1936] was to demonstrate rigorously that several fundamental problems in logic cannot be solved by algorithms at all. Complexity theory aims to show similar results for many more problems in the presence of resource limits. Such an unsolvability result, although "bad news" in most contexts (cryptographic security is an exception), can have practical benefits: it may lead to alternative models, goals, or solution strategies. As will come out in Chapters 33–34, complexity theory has so far been much more successful at drawing relative conclusions and relationships between problems than in proving absolute statements about (un)solvability.

2 Computational Models

Throughout this chapter, Σ denotes a finite alphabet of symbols; unless otherwise specified, $\Sigma = \{0,1\}$. Then Σ^* denotes the set of all finite strings, including the empty string ε, over Σ. A **language** over Σ is simply a subset of Σ^*.

We use **regular expressions** in specifying some languages. In advance of the formal definition to come in Chapter 25, we define them as one kind of "patterns" for strings to "match." The basic patterns are ε and the characters in Σ, which match only themselves. The null pattern $\emptyset$ matches no strings. Two patterns joined by "$+$" match any string that matches either of them. Two or more patterns written in sequence match any string composed of substrings that match the respective patterns. A pattern followed by a "$*$" matches any string that can be divided into zero or more successive substrings, each of which matches the pattern—here the "zero" case applies to ε, which matches any starred pattern. For example, the pattern $0(0+1)^*1$ matches any string that begins with a 0 and ends with a 1, with zero or more binary characters in between. The pattern can be used as a *name* for the language of strings that match it. In like manner, the pattern $(0^*10^*1)^*0^*$ stands for the language of binary strings that have an even number of 1s. This pattern *says* that trailing 0s are immaterial to any such string, and the rest of the string can be broken into zero or more substrings, each of which ends in a 1 and has exactly one previous 1.

2.1 Finite Automata

The finite automaton (in its deterministic version) was introduced by McCulloch and Pitts in 1943 as a logical model for the behavior of neural systems [McCulloch and Pitts, 1943]. Rabin and Scott introduced the nondeterministic finite automaton (NFA) in [Rabin and Scott, 1959], and showed that NFAs are equivalent to deterministic finite automata, in the sense that they recognize the same class of languages. This class of languages, called the **regular languages**, had already been characterized by Kleene [Kleene, 1956] and Chomsky and Miller [Chomsky and Miller, 1958] in terms of regular expressions and regular grammars, which will be described formally in Chapter 25.

In addition to their original role in the study of neural nets, finite automata have enjoyed great success in many fields such as the design and analysis of sequential circuits

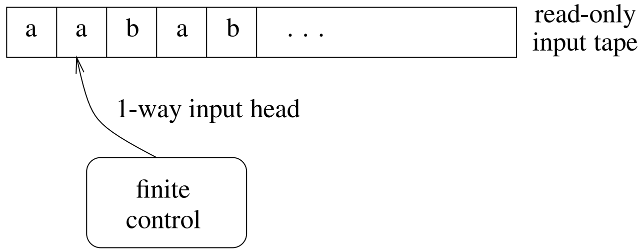


Figure 1: A finite automaton.

[Kohavi, 1978], asynchronous circuits [Brzozowski and Seger, 1994], text-processing systems [Lesk, 1975], and compilers. They also led to the design of more efficient algorithms. One excellent example is the development of linear-time string-matching algorithms, as described in [Knuth *et al*, 1977]. Other applications of finite automata can be found in computational biology [Searls, 1993], natural language processing, and distributed computing [Lynch, 1996].

A **finite automaton**, pictured in Figure 1, consists of an *input tape* and a *finite control*. The input tape contains a finite string of input symbols, and is read one symbol at a time from left to right. The finite control is connected to an *input head* that reads each symbol, and can be in one of a finite number of *states*. The input head is *one-way*, meaning that it cannot "backspace" to the left, and *read-only*, meaning that it cannot alter the tape. At each *step*, the finite control may change its state according to its current state and the symbol read, and the head advances to the next tape cell. In an NFA there may be more than one choice of next state in a step. Figure 1 also shows the second step of a computation on an input string beginning *aabab*... When the input head reaches the right end of the input tape, if the machine is in a state designated "final" (or "accepting"), we say that the input string is *accepted*; else we say it is *rejected*. The following is the formal definition.

DEFINITION 2.1 A **nondeterministic finite automaton** (NFA) is a quintuple (Q,Σ,δ,q_0,F), where

- Q is a finite set of *states*;
- Σ is a finite set of *input symbols*;
- δ, the *state transition function*, is a mapping from $Q \times \Sigma$ to subsets of Q;
- $q_0 \in Q$ is the *start state* of the NFA;
- $F \subseteq Q$ is the set of *final states*.

If δ maps $|Q| \times \Sigma$ to singleton subsets of Q, then we call such a machine a **deterministic finite automaton** (DFA).

Note that a DFA is treatable as a special case of an NFA, where the next state is always uniquely determined by the current state and the symbol read. On any input string $x \in \Sigma^*$, a DFA follows a unique *computation path*, starting in state q_0. If the final state in the path is in F, then x is *accepted*, and the path is an *accepting path*. An NFA, however, may have multiple computation paths on the same input x. It is useful to imagine that when an NFA has more than one next state, all options are taken in parallel, so that the "super-computation" is a tree of branching computation paths. Then the NFA is said to *accept x* if at least *one* of those paths is an accepting path.

REMARKS: The concept of a nondeterministic automaton, and especially the notion of acceptance, can be nonintuitive and confusing at first. We can, however, explain them in terms that should be familiar, namely those of a *solitaire* game such as "Klondike." The game starts with a certain arrangement of cards, which we can regard as the input, and has a desired "final" position—in Klondike, when all the cards have been built up by suit from ace to king. At each step, the rules of the game dictate how a new arrangement of cards can be reached from the current one—and the element of nondeterminism is that there is often more than one choice of move (otherwise the game would be little fun!). Some positions have no possible move, and lose the game. Most crucially, some positions have moves that unavoidably lead to a loss, and other moves that keep open the possibility of winning. Now for a given position, the important analytical question is, "Is there a way to win?" The answer is yes so long as there is at least one sequence of moves that ends in a (or the) desired final position. For the starting position, this condition is much the same as for an input to be accepted by an NFA. (Practically speaking, some winnable starting positions may give so many chances to go wrong along the way that a player may have little chance to find a winning sequence of moves. That, however, is beside the point in defining which positions are *winnable*. If one can always (efficiently!) answer the yes/no question of whether a given position is winnable, then one can always avoid losing moves—and win—so long as the start position is winnable to begin with.)

In any event, the set of strings accepted by a (deterministic or nondeterministic) finite automaton M is denoted by $L(M)$. When we say that *M accepts a language L*, we mean that M accepts all strings in L *and nothing else*, i.e. $L = L(M)$. Two machines are **equivalent** if they accept the same language.

Nondeterminism is capable of modeling many important situations other than solitaire games. Concurrent computing offers some examples. Suppose a device or resource (such as a printer or a network interface) is controllable by more than one process. Each process could change the state of the device in a different way. Since there may be no way to predict the order in which processes may be given control in any step, the evolution of the device may best be regarded as nondeterministic.

Sometimes a state change can occur without input stimulus. This can be modeled by allowing an NFA to make ε-*transitions*, which change state without advancing the read head. Then the second argument of δ can be ε instead of a character in Σ, and these transitions can even be nondeterministic, for instance if $\delta(q_0, \varepsilon) = \{q_1, q_2\}$. It is not hard to see that by suitable "lookahead" on states reachable by ε-*transitions*, one can always convert such

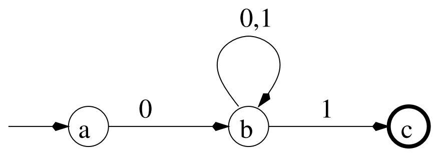


Figure 2: An NFA accepting $0(0+1)^*1$.

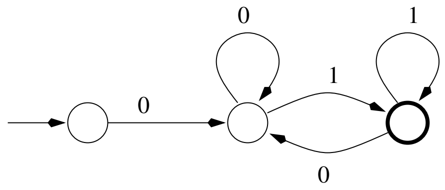


Figure 3: A DFA accepting $0(0+1)^*1$.

a machine into an equivalent NFA that does not use them.

EXAMPLE 2.1 We design an NFA to accept the language $0(0+1)^*1$. Recall that this regular expression defines those strings in $\{0,1\}^*$ that begin with a 0 and end with a 1. A standard way to draw finite automata is exemplified by Figure 2. As a convention, each circle represents a state, and an unlabeled arrow points to the start state (here, state "a"). Final states such as "c" have darker (or double) circles. The transition function δ is represented by the labeled edges. For example, $\delta(\mathrm{a}, 0) = \{\mathrm{b}\}$, and $\delta(\mathrm{b}, 1) = \{\mathrm{b}, \mathrm{c}\}$. When a transition is missing, such as on input 1 from state "a" and on both inputs from state "c", the transition is assumed to lead to an implicit non-accepting "dead" state, which has transitions to itself on all inputs. In a DFA such a dead state must be included when counting the number of states, while in an NFA it can be left out.

The machine in Figure 2 is nondeterministic since from "b" on input 1 the machine has two choices: stay at "b" or go to "c". Figure 3 gives a DFA that accepts the same language. The DFA has four, not three, states, since a dead state reached by an initial '1' is not shown.

EXAMPLE 2.2 The DFA in Figure 4 accepts the language of all strings in $\{0,1\}^*$ with an even number of 1's, which has the regular expression $(0^*10^*1)^*0^*$.

EXAMPLE 2.3 For a final example of a regular language, we introduce a general "tiling problem" to be discussed further in Chapter 26, and then strip it down to a simpler problem. A *tile* is a unit-sized square divided into four quarters by joining two diagonals. Each quarter has a color chosen from a finite set C of colors. Suppose you are given a set T of different

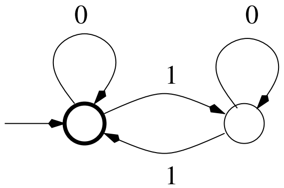


Figure 4: A DFA accepting $(0^*10^*1)^*0^*$.

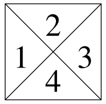


Figure 5: Numbering the quarters of a tile.

types of tiles, and have an unlimited supply of each type. A $k \times n$ rectangle is said to be *tiled* using the tiles in T if the rectangle can be filled with exactly kn unit tiles (with no overlaps) such that at every edge between two tiles, the quarters of the two tiles sharing that edge have the same color. The tile set T is said to *tile* an entire plane if the plane can be covered with tiles subject to the color compatibility stated above. As a standard application of König's infinity lemma (for which see [Knuth, 1969], Chapter 2, p. 381), it can be shown that the entire plane can be tiled with a tile set T if and only if all finite integer sided rectangles can be tiled with T. We will see in Chapter 26 that the problem of whether a given tile set T can tile the entire plane is *unsolvable*. Chapter 26 will also say more about the meaning and implications of this tiling problem.

Here we will consider a simpler problem: Let k be a fixed positive integer. Given a set T of unit tiles, we want to know whether T can tile an infinite *strip* of width k. The answer is yes if T can tile any $k \times n$ rectangle for all n. It turns out that this problem is solvable by an efficient algorithm. One way to design such an algorithm is based on finite automata. We present the solution for $k = 1$ and leave the generalization for other values of k as an exercise. Number the quarters of each tile as in Figure 5. Given a tile set T, we want to know whether for all n, the $1 \times n$ rectangle can be tiled using T.

To use finite automata, we define a language that corresponds to valid tilings. Define Σ, the input alphabet, to be T, the tile set. Each tile in T can be described by a 4-tuple $[A,B,P,Q]$ where A, B, P, and Q are (possibly equal) members of the color set C. Next we define a language L over Σ to be the set of strings $T_1T_2...T_n$ such that (i) each T_i is in Σ, and (ii) for each i, $1 \le i \le n-1$, T_i's third-quadrant color is the same as T_{i+1}'s first-quadrant color. These two conditions say that $T_1T_2...T_n$ is a valid $1 \times n$ tiling.

We will now informally describe a DFA M_L that recognizes the language L. Basically, M_L "remembers" (using the current state as the memory) the relevant information—for this

problem, it need only remember the third-quadrant color Q of the *most recently seen* tile. Suppose the DFA's current state is Q. If the next (input) tile is $[X,Y,W,Z]$, it is consistent with the last tile if and only if $Q = X$. In this case, the next state will be W. Otherwise, the tile sequence is inconsistent, so M_L enters a "dead state" from which it can never leave and rejects. All other states of M_L are accepting states. Then the infinite strip of width 1 can be tiled if and only if the language L accepted by M_L contains strings of all lengths. There are standard algorithms to determine this property for a given DFA.

The next three theorems show the satisfying result that all the following language classes are identical:

- the class of languages accepted by DFAs;
- the class of languages accepted by NFAs;
- the class of languages generated by regular expressions.
- the class of languages generated by the right-linear, or Type-3, grammars, which are formally defined in Chapter 25 and informally used here.

This class of languages is called the **regular languages**.

THEOREM 2.1 *For every NFA, there is an equivalent DFA.*

Proof. An NFA might look more powerful since it can carry out its computation in parallel with its nondeterministic branches. But since we are working with a *finite number* of states, we can simulate an NFA $M = (Q,\Sigma,\delta,q_0,F)$ by a DFA $M' = (Q',\Sigma,\delta',q_0',F')$, where

- $Q' = \{[S] \,:\, S \subseteq Q\}$;
- $q_0' = [\{q_0\}]$;
- $\delta'([S],a) = [S'] = [\cup_{q_l \in S}\delta(q_l,a)]$;
- F' is the set of all subsets of Q containing a state in F.

Here square brackets have been placed around sets of states to help one view these sets as being states of the DFA M'. The idea is that whenever M' has read some initial segment y of its input x, its current state equals the set of states q such that M has a computation path reading y that leads to state q. When all of x is read, this means that M' is in an accepting state if and only if M has an accepting computation path. Hence $L(M) = L(M')$. □

EXAMPLE 2.4 Example 2.1 contains an NFA and an equivalent DFA accepting the same language. In fact the above proof provides an effective procedure for converting an NFA to a DFA. Although each NFA can be converted to an equivalent DFA, the resulting DFA may require exponentially many more states, since the above procedure may assign a state for every eligible subset of the states of the NFA. For any $k > 0$, consider the language $L_k =$

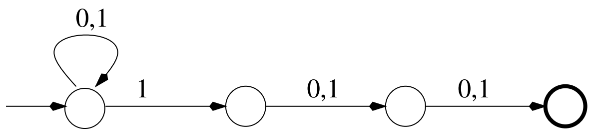


Figure 6: An NFA accepting L_3.

$\{x \mid x \in \{0,1\}^*$ and the kth letter from the right of x is a $1\}$. An NFA of $k+1$ states (for $k=3$) accepting L_k is given in Figure 6. Now we claim that any DFA M accepting L_k needs a separate state for every possible value $y \in \{0,1\}^k$ of the last k bits read. Take any distinct $y_1, y_2 \in \{0,1\}^k$ and let i be a position in which they differ. Let $z = 0^{k-i}$. Then M must accept one of the strings y_1z, y_2z and reject the other. This is possible only if the state M is in after processing y_1 (with z to come) is different from that after y_2, and thus M needs a different state for each string in $\{0,1\}^k$. The 2^k required states are also sufficient, as the reader may verify.

The remaining results of this section point forward to the formal-language models defined in Chapter 25. The point of including them here is to show the power of the finite automaton model. Regular expressions have been defined above, while a *regular grammar* over Σ consists of a set V of variable symbols, a starting variable $S \in V$, and a set P of substring-rewrite rules of the forms $A \to cB$, $A \to \varepsilon$, or $A \to c$, where $A, B \in V$ and $c \in \Sigma$.

THEOREM 2.2 *A language L is generated by a regular grammar if and only if L is accepted by an NFA.*

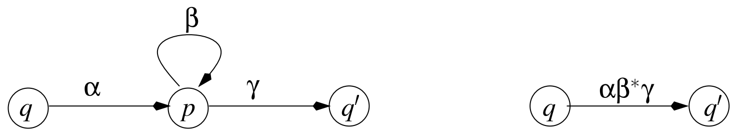
Proof. Let L be accepted by an NFA $M = (Q, \Sigma, \delta, q_0, F)$. We define an equivalent regular grammar $G = (\Sigma, V, S, P)$ by taking $V = Q$ with $S = q_0$, adding a rule $q_i \to cq_j$ whenever $q_j \in \delta(q_i, c)$, and adding rules $q_j \to \varepsilon$ for all $q_j \in F$. Then the grammar simulates computations by the NFA in a direct manner, giving $L(G) = L(M)$.

Conversely, suppose L is the language of a regular grammar $G = (\Sigma, V, S, P)$. We design an NFA $M = (Q, \Sigma, \delta, q_0, F)$ by taking $Q = V \cup \{f\}$, $q_0 = S$, and $F = \{f\}$. To define the δ function, we have $B \in \delta(A, c)$ iff $A \to cB$. For rules $A \to c$, $\delta(A, c) = \{f\}$. Then $L(M) = L(G)$—if this is not clear already, the treatment of grammars in Chapter 25 will make it so. □

THEOREM 2.3 *A language L is specified by a regular expression if and only if L is accepted by an NFA.*

PROOF SKETCH. *Part 1.* We inductively convert a regular expression to an NFA that accepts the same language as follows.

- The pattern ε converts to the NFA $M_\varepsilon = (\{q\}, \Sigma, \emptyset, q, \{q\})$, which accepts only the empty string.



(a) (b)

Figure 7: Converting an NFA to a regular expression.

- The pattern $\emptyset$ converts to the NFA $M_\emptyset = (\{q\}, \Sigma, \emptyset, q, \emptyset)$, which accepts no strings at all.
- For each $c \in \Sigma$, the pattern c converts to the NFA $M_c = (\{q, f\}, \Sigma, \delta(q, c) = \{f\}, q, \{f\})$, which accepts only the string c.
- A pattern of the form $\alpha + \beta$, where α and β are regular expressions that (by induction hypothesis) have corresponding NFAs M_α and M_β, converts to an NFA M that connects M_α and M_β in parallel: M includes all the states and transitions of M_α and M_β and has an extra start state q_0, which is connected by ε-transitions to the start states of M_α and M_β.
- A pattern of the form $\alpha \cdot \beta$, where α and β have the corresponding NFAs M_α and M_β, converts to an NFA M that connects M_α and M_β in series: M includes all the states and transitions of M_α and M_β and has extra ε-transitions from the final states of M_α to the start state of M_β. The start state of M is that of M_α, while the final states of M are those of M_β.
- A pattern of the form α^*, where α has the corresponding NFA M_α, converts to an NFA M that figuratively feeds M_α back into itself. M includes the states and transitions of M_α, plus ε-transitions from the final states of M_α back to its start state. This state is not only the start state of M but the only final state of M as well.

Part 2. We now show how to convert an NFA to an equivalent regular expression. The idea used here is based on [Brzozowski and McCluskey, 1963]; see also [Brzozowski and Seger, 1994] and [Wood, 1987].

Given an NFA M, add a new final state t, and add ε-transitions from each old final state of M to t. Also add a new start state s with an ε-transition to the old start state of M. The idea is to eliminate all states p other than s and t as follows. To eliminate a state p, we eliminate each arc coming in to p from some other state q as follows: For each triple of states q, p, q' as shown in Figure 7(a), add the transition(s) shown in Figure 7(b). (Note that if p does not have a transition leading back to p, then $\beta = \beta^* = \varepsilon$.) After we have considered all such triples, we can delete state p and transitions related to p. Finally, we obtain Figure 8, and the final α is a regular expression for $L(M)$. □

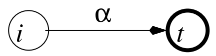


Figure 8: The reduced NFA.

The last three theorems underline the importance of the class of regular languages, since it connects to notions of automata, grammars, patterns, and much else. However, regular languages and finite automata are not powerful enough to serve as our model for a modern computer. Many extremely simple languages cannot be accepted by DFAs. For example, $L = \{xx \mid x \in \{0,1\}^*\}$ cannot be accepted by a DFA. To show this, we can argue similarly to the "L_k" languages in Example 2.4 that for any two strings $y_1 = 0^m1$ and $y_2 = 0^n1$ with $n \neq m$, a hypothetical DFA M would need to be in a different state after processing y_1 from that after y_2, because with $z = 0^m1$ it would need to accept y_1z and reject y_2z. However, in this case we would conclude that M needs *infinitely* many states, contradicting the definition of a *finite* automaton. Hence the language L is not regular. Other ways to prove assertions of this kind include so-called "pumping lemmas" or a direct argument that some strings y contain more information than a *finite* state machine can *remember* [Li and Vitányi, 1993]. We refer the interested readers to Chapter 25 and textbooks such as [Hopcroft and Ullman, 1979], [Gurari, 1989], [Wood, 1987], and [Floyd and Beigel, 1994].

One can also try to generalize the DFA to allow the input head to move backwards as well as forwards, in order to review earlier parts of the input string, while keeping it read-only. However, such "two-way DFAs" are not more powerful—they can be simulated by normal DFAs. The point of departure for a more-powerful model is to allow the machine to *write* on its tape and later review what it has written. Then the tape becomes a *storage* medium, not just a sequence of events to react to. This ability to write down intermediate results for future reference makes DFAs into full-blown general-purpose computers.

2.2 Turing Machines

A **Turing machine** (TM), pictured in Figure 9, consists of a *finite control*, an infinite *tape* divided into cells, and a read/write *head* on the tape. The finite control can be in any one of a finite set Q of states. Each cell can contain one symbol from the *tape alphabet* Γ, which contains the *input alphabet* Σ and a special character B called the *blank*. Γ may contain other characters besides B and those in Σ, but most often $\Sigma = \{0,1\}$ and $\Gamma = \{0,1,B\}$. We refer to the two directions on the tape as *left* and *right*, and abbreviate them by L and R. At any time, the head is positioned over a particular cell that it is said to *scan*. Initially, the head scans a distinguished cell on the tape called the *start cell*, the finite control is in the start state q_0, and all cells contain B except for a contiguous finite sequence of cells, extending from the start cell to the right, that contain characters in Σ. These cells hold the *input string* x; in case $x = \varepsilon$ the whole tape is blank. The machine is said to begin its computation on input x at time 0, and computation unfolds in discrete time steps numbered $1, 2, \ldots$

In any step, contingent on its current state and the character being scanned, the device is allowed to perform one the following two basic operations:

1. write a symbol from the tape alphabet Γ into the scanned cell, or

2. shift the head one cell left or right.

Then it may change its internal state in the same step. The allowed actions of a particular machine are specified by a finite set δ of *instructions*. Each instruction has the form (q,c,d,r) with $q,r \in Q$, $c \in \Gamma$, and either $d \in \Gamma$ or $d \in \{L,R\}$. This means that if the machine is in state q scanning character c on the tape, it may either change c to d (if $d \in \Gamma$) or move its head (if $d \in \{L,R\}$), and it enters state r. Either $c = d$ or $q = r$ is allowed. (Many texts use an alternate formalism in which *both* basic operations may be performed in the same step, so that instructions have the form (q,c,d,D,r) with $q,r \in Q$, $c,d \in \Gamma$, and $D \in \{L,R\}$, sometimes adding the option $D = S$ of keeping the head stationary. The differences do not matter for our purposes.)

If for every combination of q and c there is at most one instruction (q,c,d,r) that the machine can execute, then the machine is *deterministic*. Otherwise, the machine is *nondeterministic*. In order for computations to possibly halt, there must be some combinations q,c for which δ has *no* instruction (q,c,d,r). If a computation reaches such a state q while scanning c, the device is said to *halt*. Then if q is designated as a final state, we say the machine *accepts* its input string x; if q is not a final state, we say that the machine *rejects* the input. We adopt the convention that there is only one final state labeled q_f, and that q_f is also a halting state, meaning that there is no instruction (q,c,d,r) with source state $q = q_f$ at all.

DEFINITION 2.2 A **Turing machine** is a 7-tuple $M = (Q,\Sigma,\Gamma,\delta,B,q_0,q_f)$, where each of the components has been described above.

Given an input x, a deterministic Turing machine M carries out a uniquely determined succession of operations, which may or may not terminate in a finite number of steps. If it terminates, then the *output* $M(x)$ is determined to be the longest string of characters over Σ beginning in the cell in which the head halted and extending to the right. (If the scanned

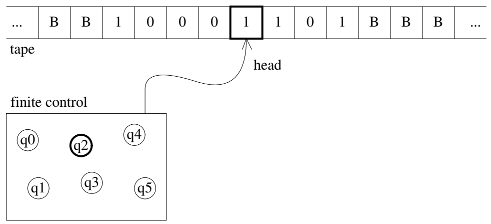


Figure 9: A Turing machine.

cell holds B or some other character in $\Gamma \setminus \Sigma$, then the output is ε.) A function $f : \Sigma^* \to \Sigma^*$ is **computable** if there is a Turing machine M such that for all inputs $x \in \Sigma^*$, $M(x) = f(x)$.

A nondeterministic Turing machine is analogous to an NFA. One may imagine that it carries out its computation in parallel. The computation may be viewed as a (possibly infinite) tree, each of whose nodes is labeled by a configuration of the machine. A *configuration* specifies the current state q, the position of the tape head, and the contents of the tape, including the character c currently being scanned. Each node has as many children as there are different instructions $(q, c, \cdot, \cdot)$ to execute, and each child is the configuration that results from executing the corresponding instruction. The root of the tree is the starting configuration of the machine. If any of the branches terminates in the final state q_f, we say the machine accepts the input. Note that this is the same "benefit of all doubt" criterion for acceptance that we discussed above for NFAs. Note also that a deterministic Turing machine always defines a "tree" with a single branch that forms a simple (possibly infinite) path.

A language L is said to be **recursively enumerable (r.e.)** if there is a Turing machine M such that $L = \{x : M \text{ accepts } x\}$. Furthermore, if M is *total*; i.e., if every computation of M on every input x halts, then $L(M)$ is **recursive**. These terms reflect an important correspondence between languages and functions. For any language $L \subseteq \Sigma^*$, define the *characteristic function* f_L by $f_L(x) = 1$ of $x \in L$, $f(x) = 0$ otherwise. Then a language L is recursive if and only if f_L is computable—and computable functions were originally defined with regard to formalisms that used *recursion*. Note that having L be acceptable by a Turing machine M is *not* enough for f_L to be computable, because there may be inputs $x \notin L$ for which the computation of M on x never halts.

Conversely, given a function $f : \Sigma^* \to \Sigma^*$, and letting # be a new input symbol not in Σ, one can define the language $L_f = \{x\#y : y \text{ is an initial segment of } f(x)\}$. Recognizing L_f allows one to find successive bits of the value $f(x)$. Hence it is common in the field to identify function problems with language problems and concentrate on the latter. A language can also be identified with a property of strings and with the associated **decision problem** "given a string x, does x have the property?" For instance, the problem of deciding whether a given number is prime is identifiable with the language of (binary string encodings of) prime numbers. The problem is **decidable** if the associated language is recursive, and a total Turing machine accepting the language is said to *decide* the problem. The term "decidable language" is a synonym for "recursive language," and "recursive function" is a synonym for "computable function." A Turing machine M that does not halt on all inputs computes a **partial recursive** function (whose domain is a proper subset of Σ^*), and $L(M)$ is a **partially decidable** language (or problem, or property). Any problem or language that is not decidable by a Turing machine is called **undecidable**, and any (partial or total) function that is not computable by a Turing machine is called **uncomputable**.

Now when we say that a Turing machine M' *simulates* another Turing machine M, we usually mean more than saying they accept the same language or compute the same (partial) function. Usually there is some overt correspondence between computations of M and those of M'. This is so in the simulations claimed by the following theorem, which says that many variations in the basic machine model do not alter the notion of computability.

THEOREM 2.4 *All the following generalizations of Turing machines can be simulated by the one-tape deterministic Turing machine model defined in Definition 2.2, with tape alphabet* $\{0, 1, B\}$.

- *enlarging the tape alphabet* Γ*;*
- *adding more tapes;*
- *adding more read/write heads or other access points on each tape;*
- *having two- or higher-dimensional grids in place of tapes, where the head may move to any adjacent grid cell;*
- *allowing nondeterminism.*

Extra tapes after the input tape are called *worktapes* provided they allow read-write access. A two-tape Turing machine, or alternately a one-tape machine with two heads, has instructions with six components: current state, two characters read by the heads, two head actions, next state. Although these generalizations do not make a Turing machine compute more, they do make Turing machines more efficient and easier to program. Many more variants of Turing machines have been studied and used in the literature.

Of all the simulations in Theorem 2.4, the last one needs special comment. A nondeterministic computation branches like a tree. The easiest way for a deterministic Turing machine to simulate it is by traversing the tree in a breadth-first manner, which is the same as trying all possibilities at any step with nondeterministic choices. However, even if there are at most two choices at any step, simulating n steps of the NTM could take on the order of 2^n steps by the DTM. Whether a more-efficient simulation is possible is bound up with the famous "P vs. NP" problem, to be discussed below and in Chapter 33.

EXAMPLE 2.5 A DFA can be regarded as the special case of a Turing machine in which every instruction moves the head right. Turing machines naturally accept more languages than DFAs can. For example, a Turing machine can accept the non-regular language $L = \{xx \mid x \in \{0,1\}^*\}$ as follows. Given an input string $w \in \{0,1\}^*$:

- First find the middle point. A TM with two tape heads can do this efficiently by moving one head twice for every move of the other, until the further-advanced head sees the blank that marks the end of w. This stage can also tell whether w has even or odd length and immediately *reject* in the latter case.
- Then check whether the scanned characters match while moving both heads one cell left until the leftmost head sees the blank to the left of the beginning of w. If all pairs match, *accept*, else *reject*.

For a TM with only one head the strategy is more cumbersome. One way is to use "alias" characters a for 0 and b for 1, aliasing first the leftmost 0/1 character on the tape, then the rightmost, then the next-leftmost,... until finding the character just left of middle (if w has even length). Then "un-alias" it and check that the rightmost aliased character matches it, un-aliasing the latter as well. By looking for cases of an a or b immediately to the left of an un-aliased 0 or 1, the TM can repeat this check until all of the left half is compared with the right half. Whereas the two-head TM needs only $3n/2$ steps to decide whether a string w of length n belongs to L, the one-head TM takes about n^2 steps. It is known that n^2 steps are necessary (asymptotically) for any one-head TM to accept the language L (see, for instance, [Hopcroft and Ullman, 1979] or [Li and Vitányi, 1993]).

Three restrictions of the notion of a Turing machine tape merit special mention.

- A *pushdown store* (or *stack*) is a semi-infinite worktape with one head such that each time the head moves to the left, it erases the symbol scanned previously. This is a last-in first-out storage.
- A *queue* is a semi-infinite work tape with two heads that only move to the right, the leading head is write-only and the trailing head is read-only. This is a first-in first-out storage.
- A *counter* is a pushdown store with a single-letter alphabet, except for a special bottom-of-stack marker that allows testing the counter for zero. Then a push increments the counter by one, and a pop decrements it by one.

EXAMPLE 2.6 A **pushdown automaton** (PDA) has one read-only input tape and one pushdown store. A PDA can be identified with a two-tape Turing machine whose tape-1 head can never move left and whose tape-2 head can move left only while scanning a blank (combined with a previous step that writes a blank, this simulates popping the stack).

Pushdown automata have been thoroughly studied because nondeterministic PDAs accept precisely the class of context-free languages, to be defined in Chapter 25. Various types of PDAs have fundamental applications in compiler design.

The PDA has less power than a Turing machine. For example, $L = \{xx \mid x \in \{0,1\}^*\}$ cannot be accepted by a PDA, but it can be accepted by a Turing machine as in Example 2.5. However, a PDA is more powerful than a DFA. For example, a PDA can accept the non-regular language $L' = \{0^n 1^n \mid n \geq 0\}$ easily: For each initial 0 read, push a 0 onto the stack; then pop a 0 for each 1 read, and accept if and only if a blank is read after the block of 1s exactly when the stack is empty. Indeed, this PDA is a counter machine with a single counter.

Two pushdown stores can easily be used to simulate a tape—let one stack represent the part of the tape currently to the left of the input head, and let the other stack represent the rightward portion. Much more subtle is the fact that two *counters* can simulate a tape; unlike the two-pushdown case this takes exponentially more time. Finally, a single queue can simulate a tape: send the lead head to the right end so that it can write the next-step update of the configuration that the trailing head is reading. This involves encoding the current state of the TM being simulated onto the tape of the queue machine that is simulating it. Hence a single-queue machine, with the input initially resting in the queue, is as powerful as a Turing machine, although it may require the square of the running time. For comparisons of powers of pushdown stores, queues, counters, and tapes, see [van Emde Boas, 1990] and [Li and Vitányi, 1993].

Much more important is the fact that there are single Turing machines that are capable of simulating *any* Turing machine. Formally, a **universal Turing machine** U takes an encoding $\langle M, w\rangle$ of a (deterministic) Turing machine M and a string w as input, and simulates M on input w. U accepts $\langle M, w\rangle$ if and only if M accepts w. Intuitively, U models a general-purpose computer that takes a "program" M and "data" w, and executes M on input w. Universal Turing machines have many applications. For example, the definition of Kolmogorov complexity (see [Li and Vitányi, 1993]) fundamentally relies on them.

EXAMPLE 2.7 Let $L_u = \{\langle M, w\rangle \mid M \text{ accepts } w\}$. Then L_u is *the* language accepted by a universal Turing machine, so it is recursively enumerable. We shall see in Chapter 26, however, that L_u is not recursive. The same properties hold for the language $L_h = \{\langle M, w\rangle : M \text{ on input } w \text{ halts}\}$, which is the language of the so-called **Halting Problem**.

2.3 Oracle Turing Machines

In order to study the comparative hardness of computational problems, we often need to extend the power of Turing machines by adding oracles to them.

Informally, a Turing machine T with an *oracle* A operates similarly to a normal Turing machine, with the exception that it can write down a string z and ask whether z is in the language A. The machine gets the correct yes/no answer from the oracle in one step, and can branch its computation accordingly. This feature can be used as often as desired. We now give the definition precisely.

DEFINITION 2.3 An **oracle Turing machine** is a normal Turing machine T with an extra *oracle query tape*, a special state $q_?$, and two distinguished states labeled q_y and q_n. Let A be any language over an alphabet Σ. Whenever T enters state $q_?$ with some string $z \in \Sigma^*$ on the query tape, control passes to state q_y if $z \in A$, or to q_n if $z \notin A$. The computation continues normally until the next time the machine enters $q_?$. The machine T with a given choice of oracle A is denoted by T^A.

EXAMPLE 2.8 In Example 2.7, we know that the universal language $L_u = \{\langle M, w\rangle \mid M \text{ accepts } w\}$ is not Turing decidable. But if we can use L_u as the oracle set, there is a trivial oracle TM T such that T with oracle L_u decides L_u. T simply copies its input x onto the query tape and enters $q_?$. If control passes to q_y, T accepts; otherwise from q_n it rejects.

For something less trivial, suppose we have $L_h = \{\langle M, w\rangle : M \text{ on input } w \text{ halts}\}$ as the oracle set. Given a (non-oracle) Turing machine M, there is a standard way to modify M to the code of an equivalent Turing machine M' in which the accepting state q_f is the only place where M' can halt. This is done by making every other combination q, c where M

might halt send control to an extra state that causes an infinite loop. Thus M' halts on w if and only if M accepts w. Now design an oracle Turing machine T' that on any input x of the form $\langle M, w\rangle$ (rejecting if x does not have the form) writes $x' = \langle M', w\rangle$ on the query tape and enters $q_?$, accepting if control goes to q_y and rejecting from q_n. Then T' with oracle set L_h decides whether $x \in L_u$, since $x \in L_u \iff x' \in L_h$. This is a simple case where an oracle for one problem helps one decide a different problem.

A language A is **Turing-reducible** to a language B, written $A \leq_T B$, if there is an oracle Turing machine that with oracle B decides A. For example, we have just shown that $L_u \leq_T L_h$. The important special case in which the oracle TM T makes exactly one query, accepting from q_y and rejecting from q_n, gets its own notation: $A \leq_m B$. Equivalently, $A \leq_m B$ if there is a computable function f such that for all $x \in \Sigma^*$, $x \in A \iff f(x) \in B$. The function f represents the computation done by T prior to making the sole query. This case is called a **many-one reduction** (hence the subscript "m") for the arcane historical reason that f need not be a one-to-one function. The term "many-one reduction" is standard now. The above example actually shows that $L_u \leq_m L_h$. It is not hard to show that also $L_h \leq_m L_u$, so that the Halting Problem and the membership problem for a universal Turing machine are *many-one equivalent*.

2.4 Alternating Turing Machines

Turing machines can be naturally generalized to model parallel computation. A nondeterministic Turing machine accepts an input if there exists a move sequence leading to acceptance. We can call any nondeterministic state entered along this sequence an *existential state*. We can naturally add another type of state, a *universal state*. When a machine enters a universal state, the machine will accept if and only if *all* moves from this state lead to acceptance. These machines are called **alternating Turing machines**.

Let us describe the computation of alternating Turing machines formally and precisely. An alternating Turing machine is simply a nondeterministic Turing machine with the extra power that some states can be universal. A *configuration* of an alternating Turing machine A has the same form as was described for a deterministic Turing machine, namely:

```
(current state, tape contents, head positions).
```

We write

$$\alpha \vdash \beta$$

if, in one step, A can move from configuration α to configuration β. A configuration with current state q is *accepting* if

- q is an accepting state (i.e., $q = q_f$); or
- q is existential, and there exists an accepting configuration β such that $\alpha \vdash \beta$; or
- q is universal, and or each configuration β such that $\alpha \vdash \beta$, β is an accepting configuration.

This definition may seem circular, but by working backwards from configurations with q_f in the current-state field, one may verify that it inductively defines the set of accepting configurations in a natural manner. Then A accepts an input x if its initial configuration (with current state q_0, x on the input tape, heads at initial positions) is accepting.

Alternating Turing machines were first proposed by [Chandra, *et al*, 1981] for the purpose of modeling parallel computation. In order to allow sub-linear computation times, a random-access model is used to read the input. When in a special "read" state, the alternating Turing machine is allowed to write a number in binary which is then interpreted as the address of a location on the input tape, whose symbol is then read in unit time. By using universal states to relate different branches of the computation, one can effectively read the whole input in as little as logarithmic time.

Just as a nondeterministic Turing machine is a model for solitaire games, an alternating Turing machine is a model for general two-person games. Alternating Turing machines have been successfully used to provide a theoretical foundation of parallel computation as well as to establish the complexity of various two person games. For example, a chess position with White to move can be modeled from White's point of view as a configuration α whose first component is an existential state. The position is winning if there exists a move for White such that the resulting position β is winning. Here β with Black to move has a universal state q, and is a winning position (for White) if and only if either Black is checkmated (this is the base case $q = q_f$) or for all moves by Black to a position γ, γ is a winning position for White…Chapter 34, Sections 5–6, will demonstrate the significance of games for time and space complexity, to which we now turn.

3 Time and Space Complexity

With Turing machines, we can now formally define what we mean by time and space complexity. The formal investigation by [Hartmanis and Stearns, 1965] and [Blum, 1967] in the 1960's marked the beginning of the field of *computational complexity*.

An important point with space complexity is that a machine should be charged only for those cells it uses for calculation, and not for read-only input, which might be provided on cheaper non-writable media such as CD-ROM or accessed piecemeal over a network. Hence we modify the Turing machine of Figure 9 by making the tape containing the input read-only, and giving it one or more worktapes.

DEFINITION 3.1 Let M be a Turing machine. If for all n, every sequence of legal moves on an input x of length n halts within $t(n)$ steps, we say that M is of **time complexity** $t(n)$. Similarly, if every such sequence uses at most $s(n)$ *worktape* cells, then M is of **space complexity** $s(n)$.

THEOREM 3.1 *Fix a number $c > 0$, a space bound $s(n)$, and a time bound $t(n)$.*

(a) *Any Turing machine of $s(n)$ space complexity, using any number of tapes or grids of any dimension, can be simulated by a Turing machine with a single (one-dimensional) worktape that has space complexity $s(n)/c$.*

(b) *Any Turing machine of $t(n)$ time complexity can be simulated by a Turing machine, with the same number and kinds of worktapes, that has time complexity $n+t(n)/c$.*

The proof of these so-called *linear speed-up theorems* involves enlarging the original TM's worktape alphabet Γ to an alphabet Γ' large enough that one character in Γ' can encode a block of c consecutive characters on a tape of the original machine (see [Hopcroft and Ullman, 1979]). The extra "$n+$" in the time for part (b) is needed to read and translate the input into the "compressed" alphabet Γ'. If we think of memory in units of bits the idea that this saves space and time is illusory, but if we regard the machine with Γ' as having a larger *word size*, the savings make sense. Definition 3.1 is phrased in a way that applies also to nondeterministic and alternating TMs, and the two statements in Theorem 3.1 hold for them as well.

In Theorem 3.1, if $s(n) \geq n$, then we do not need to separate the input tape from the worktape(s). For any Turing machine M of linear space complexity, part (a) implies that we can simulate M by a one-tape TM M' that on any input x uses only the cells initially occupied by x (except for one visit to the blank cell to the right of x to tell where x ends). Then M' is called a *linear bounded automaton*.

The main import and convenience of Theorem 3.1 is that one does not need to use "$O()$-notation" to define complexity classes: space complexity $O(s(n))$ is no different from space complexity $s(n)$, and similarly for time. As we shall see, it is not always possible to reduce the number of tapes and run in the same time complexity, so researchers have settled on Turing machines with any finite number of tapes as the bench model for time complexity.

DEFINITION 3.2

- DTIME$[t(n)]$ is the class of languages accepted by multi-tape deterministic TMs in time $t(n)$;
- NTIME$[t(n)]$ is the class of languages accepted by multi-tape nondeterministic TMs in time $t(n)$;
- DSPACE$[s(n)]$ is the class of languages accepted by deterministic TMs in space $s(n)$;
- NSPACE$[s(n)]$ is the class of languages accepted by multitape nondeterministic TMs in space $O(s(n))$;
- P is the complexity class $\bigcup_{c\in\mathcal{N}}$DTIME$[n^c]$;
- NP is the complexity class $\bigcup_{c\in\mathcal{N}}$NTIME$[n^c]$;
- PSPACE is the complexity class $\bigcup_{c\in\mathcal{N}}$DSPACE$[n^c]$;
- ATIME$[s(n),t(n)]$ is the class of languages accepted by alternating Turing machines operating simultaneously in time $t(n)$ and space $s(n)$.

EXAMPLE 3.1 In Example 2.5 we demonstrated how the language $L = \{xx \mid x \in \{0,1\}^*\}$ can be decided by a Turing machine. We gave a two-head, one-tape TM running in time $3n/2$, and it is easy to design a two-tape, one-head-per-tape TM that executes the same strategy in time $2n$. Theorem 3.1 says that by using a larger tape alphabet, one can push the time down to $(1+\varepsilon)n$ for any fixed $\varepsilon > 0$. However, our basic one-tape, one-head Turing machine model can do no better than time on the order of n^2.

EXAMPLE 3.2 *Any* multi-tape, multi-head Turing machine, not just the one accepting L in the last example, can be simulated by our basic one-tape, one-head model in at most the square of the original's running time. For example, a two-tape machine M with tape alphabet Γ can be simulated by a one-tape machine M' with a Γ' large enough to encode all pairs of characters over Γ. Then M' can regard its single tape as having two "tracks," one for each tape of M. M' also needs to mark the locations of the two heads of M, one on each track—this can be facilitated by adding more characters to Γ'. Now to simulate one step of M, the one-tape machine M' must use its single head to update the computation at the two locations with the two head markers. If M runs in time $t(n)$, then the two head markers cannot be more than $t(n)$ cells apart. Thus to simulate each step by M, M' moves its head for at most $t(n)$ distance. Hence M' runs in time at most $t(n)^2$.

The simulation idea in Example 3.2 does not work if M uses two- or higher-dimensional tapes, or a more-general "random-access" storage (see the next section). However, a one-tape M' can be built to simulate it whose running time is still no worse than a polynomial in the time of M. It is important to note that our basic one-tape deterministic Turing machine is known to simulate all of the extended models we offer above and below—except nondeterministic and alternating TMs—with at most polynomial slowdown. This is a key point in taking the class P, defined as above in terms of Turing machine time, as the benchmark for which languages and functions are considered *feasibly computable* for general computation. (See Chapter 33 for more discussion of this point.) Polynomial time for nondeterministic Turing machines defines the class NP. Interestingly, polynomial space for nondeterministic TMs does equal polynomial space for deterministic TMs [Savitch, 1970], and polynomial *time* for *alternating* TMs equals the same class, namely PSPACE.

EXAMPLE 3.3 All of the basic arithmetical operations—plus, minus, times, and division—belong to P. Given two n-digit integers x and y, we can easily add or subtract them in $O(n)$ steps. We can multiply or divide them in $O(n^2)$ steps using the standard algorithms learned in school. Actually, by grouping blocks of digits in x and y and using some clever tricks one can bring the time down to $O(n^{1+\varepsilon})$ bit-operations for any desired fixed $\varepsilon > 0$, and the asymptotically fastest method known takes time $O(n \log n \log\log n)$ [Schönhage and Strassen, 1971]. Computing x^y is technically not in P because the sheer length of the output may be exponential in n, but if we measure time as a function of output length as well as input length, it is in P. However, the operation of *factoring* a number into primes, which is a kind of inverse of multiplication, is commonly believed *not* to belong to P. The language associated to the factoring function (refer to "L_f" before Theorem 2.4 above) does belong to NP.

EXAMPLE 3.4 There are many other important problems in NP that are not known to be in P. For example, consider the following "King Arthur" problem, which is equivalent to the problem called HAMILTONIAN CIRCUIT in Chapter 34. King Arthur plans to have a round table meeting. By one historical account he had 150 knights, so let $n = 150$. It is known that some pairs of knights hate each other, and some do not. King Arthur's problem is to arrange the knights around the table so that no pair of knights who sit side by side hate each other. King Arthur can solve this problem by enumerating all possible permutations of n knights. But even at $n = 150$, there are 150! permutations. All the computers in the whole world, even if they started a thousand years ago and worked non-stop, would still be going on today, having examined only a tiny fraction of the 150! permutations. However, this problem is in NP because a nondeterministic Turing machine can just guess an arrangement and verify the correctness of the solution—by checking if any two neighboring knights are enemies—in polynomial time. It is currently unknown if every problem in NP is also in P. This problem has a special property—namely, if it is in P then every problem in NP is also in P.

The following relationships are true:

$$\mathrm{P} \subseteq \mathrm{NP} \subseteq \mathrm{PSPACE}.$$

Whether or not either of the above inclusions is proper is one of the most fundamental open questions in computer science and mathematics. Research in computational complexity theory centers around these questions. The first step in working on these questions is to identify the hardest problems in NP or PSPACE.

DEFINITION 3.3 Given two languages A and B over an alphabet Σ, a function $f : \Sigma^* \to \Sigma^*$ is called a **polynomial-time many-one reduction** from A to B if

(a) f is polynomial-time computable, and

(b) for all $x \in \Sigma^*$, $x \in A$ if and only if $f(x) \in B$.

One also writes $A \leq_m^p B$.

The only change from the definition of "many-one reduction" at the end of Section 2.3 is that we have inserted "polynomial-time" before "computable." There is also a polynomial-time version of Turing reducibility as defined there, which gets the notation $A \leq_T^p B$.

DEFINITION 3.4 A language B is called **NP-complete** if

1. B is in NP;

2. for every language $A \in \mathrm{NP}$, $A \leq_m^p B$.

In this definition, if only the second item holds, then we say the language B is *NP-hard*. (Curiously, while "NP-complete" is always taken to refer by default to polynomial-time many-one reductions, "NP-hard" is usually extended to refer to polynomial-time Turing reductions.) The upshot is that if a language B is NP-complete and B is in P, then NP = P. An NP-complete language is in this sense a hardest language in the class NP. Working independently, Cook [Cook, 1971] and Levin [Levin, 1973] introduced NP-completeness, and Karp [Karp, 1972] further demonstrated its importance. PSPACE and other classes also have complete languages.

Chapters 33 and 34 of this *Handbook* develop the topics of this section in much greater detail. We also refer the interested reader to the textbooks [Yap, 1997], [Hopcroft and Ullman, 1979], [Wood, 1987], [Lewis and Papadimitriou, 1981] and [Floyd and Beigel, 1994].

4 Other Computing Models

Over the years, many alternative computing models have been proposed. Under reasonable definitions of running time for these models, they can all be simulated by Turing machines with at most a polynomial slow-down. The reference [van Emde Boas, 1990] provides a nice survey of various computing models other than Turing machines. We will discuss a few such alternatives very briefly and refer our readers to Chapter 48 and [van Emde Boas, 1990] for more information.

4.1 Random Access Machines

The *Random Access Machine* [Cook and Reckhow, 1973] consists of a finite control where a program is stored, several arithmetic registers, and an infinite collection of memory registers $R[1], R[2], \ldots$ All registers have an unbounded word length. The basic instructions for the program are LOAD, STORE, ADD, MUL, GOTO, and conditional-branch instructions. The LOAD and STORE commands can use indirect addressing. Compared to Turing machines, this appears to be a closer but more complicated approximation of modern computers. There are two standard ways for measuring time complexity of the model:

- The *Unit-cost RAM*: Here each instruction takes one unit of time, no matter how big the operands are. This measure is convenient for analyzing many algorithms.

- The *Log-cost RAM*: Here each instruction is charged for the sum of the lengths of all data manipulated by the instruction. Equivalently, each use of an integer i is charged $\log i$ time units, since $\log i$ is approximately the length of i. This is a more realistic model, but sometimes less convenient to use.

Log-cost RAMs and Turing machines can simulate each other with polynomial overheads. The unit-cost assumption becomes unrealistic when the MUL instruction is used repeatedly to form exponentially large numbers. Taking MUL out, however, makes the unit-cost RAM polynomially equivalent to the Turing machine as well.

4.2 Pointer Machines

Pointer machines were introduced by [Kolmogorov and Uspenskii, 1958] in 1958 and in modified form by Schönhage in the 1970s. Schönhage called his form the "storage modification machine" [Schönhage, 1980], and both forms are sometimes named for their authors. We informally describe Schönhage's form here here. A pointer machine is similar to a RAM, but instead of having an unbounded array of registers for its memory structure, it has modifiable pointer links that form a Δ-*structure*. A Δ-structure S, for a finite alphabet Δ of size k, is a finite directed graph in which each node has k out-edges labeled by the k symbols in Δ. Every node also has a cell holding an integer, as with a RAM. At every step of the computation, one node of S is distinguished as the *center*, which acts as a starting point for addressing. A word $w \in \Delta^*$ addresses the cell of the node formed by following the path of pointer links selected by the successive characters in w. Besides having all the RAM instructions, the pointer machine has various instructions and rules for moving its center and redirecting pointer links, thus modifying the storage structure. Under the log-cost criterion, pointer machines are polynomially equivalent to RAMs and Turing machines. There are many interesting studies on the precise efficiency of the simulations among these models, and we refer to the reader to the survey [van Emde Boas, 1990] as a center for further pointers on them.

4.3 Circuits and Nonuniform Models

A *Boolean circuit* is a finite, labeled, directed acyclic graph. Input nodes are nodes without ancestors; they are labeled with input variables $x_1, \ldots, x_n$. The internal nodes are labeled with functions from a finite set of Boolean operations such as $\{$AND, OR, NOT$\}$ or $\{$NAND$\}$. The number of ancestors of an internal node is precisely the number of arguments of the Boolean function that the node is labeled with. A node without successors is an output node. The circuit is naturally evaluated from input to output: at each node the function labeling the node is evaluated using the results of its ancestors as arguments. Two cost measures for the circuit model are

- *depth*: the length of a longest path from an input node to an output node.
- *size*: the number of nodes in the circuit.

These measures are applied to a family $\{C_n \mid n \geq 1\}$ of circuits for a particular problem, where C_n solves the problem of size n. Subject to the *uniformity condition* that the layout of C_n be computable given n (in time polynomial in n), circuits are (polynomially) equivalent to Turing machines. Chapters 33 and 48 give full presentations of circuit complexity, while [van Emde Boas, 1990] and [Karp and Ramachandran, 1990] have more details and pointers to the literature.

5 Defining Terms

alternating Turing machine: a generalization of a nondeterministic Turing machine. In the latter, every state can be called an existential state since the machine accepts if one of the possible moves leads to acceptance. In an alternating Turing machine there are also universal states, from which the machine accepts only if all possible moves out of that state lead to acceptance.
algorithm: a finite sequence of instructions that is supposed to solve a particular problem.
complexity class NP: the class of languages that can be accepted by a nondeterministic Turing machine in polynomial time.
complexity class P: the class of languages that can be accepted by a deterministic Turing machine in polynomial time.
complexity class PSPACE: the class of languages that can be accepted by a Turing machine in polynomial space.
computable function: a function that can be computed by an algorithm—equivalently, by a Turing machine.
decidable problem/language: a problem that can be decided by an algorithm—equivalently, whose associated language is accepted by a Turing machine that halts for all inputs.
deterministic: permitting at most one next move at any step in a computation.
finite automaton or finite-state machine: a restricted Turing machine where the head is read-only and shifts only from left to right.
(formal) language: a set of strings over some fixed alphabet.
Halting Problem: the problem of deciding whether a given program (or Turing machine) halts on a given input.
many-one reduction: a reduction that maps an instance of one problem into an equivalent instance of another problem.
nondeterministic: permitting more than one choice of next move at some step in a computation.
NP-complete language: a language in NP such that every language in NP can be reduced to it in polynomial time.
oracle Turing machine: a Turing machine with an extra oracle tape and three extra states $q_?, q_y, q_n$. When the machine enters $q_?$, control goes to state q_y if the oracle tape content is in the oracle set; otherwise control goes to state q_n.
partial recursive function: a partial function computed by a Turing machine that need not halt for all inputs.
partially decidable problem: one whose associated language is recursively enumerable. Equivalently, there exists a program that halts and outputs 1 for every instance having a yes answer, but is allowed not to halt or to halt and output 0 for every instance with a no answer.
polynomial time reduction: a reduction computable in polynomial time.
program: a sequence of instructions that can be executed, such as the code of a Turing machine or a sequence of RAM instructions.

pushdown automaton: a restricted Turing machine where the tape acts as a pushdown store (or a stack), with an extra one-way read-only input tape.
recursive language: a language accepted by a Turing machine that halts for all inputs.
recursively enumerable (r.e.) language: a language accepted by a Turing machine.
reduction: a computable transformation of one problem into another.
regular language: a language which can be described by some right-linear/regular grammar (or equivalently by some regular expression).
time/space complexity: a function describing the maximum time/space required by the machine on any input of length n.
Turing machine: a simplest formal model of computation consisting a finite-state control and a semi-infinite sequential tape with a read-write head. Depending on the current state and symbol read on the tape, the machine can change its state and move the head to the left or right. Unless otherwise specified, a Turing machine is deterministic.
Turing reduction: a reduction computed by an oracle Turing machine that halts for all inputs with the oracle used in the reduction.
uncomputable function: a function that cannot be computed by any algorithm—equivalently, not by any Turing machine.
undecidable problem/language: a problem that cannot be decided by any algorithm—equivalently, whose associated language cannot be recognized by a Turing machine that halts for all inputs).
universal Turing machine: a Turing machine that is capable of simulating any other Turing machine if the latter is properly encoded.

References

[Blum, 1967] Blum, M. 1967. A machine independent theory of complexity of recursive functions. *J. Assoc. Comput. Mach.* 14:322-336, 1967.

[Brzozowski and McCluskey, 1963] Brzozowski, J. and McCluskey Jr, E. 1963. Signal flow graph techniques for sequential circuit state diagram. *IEEE Trans. on Electronic Computers*, EC-12(2):67-76.

[Brzozowski and Seger, 1994] Brzozowski, J.A. and Seger, C-J. H. 1994. *Asynchronous Circuits*, Springer-Verlag, New York.

[Chandra, *et al*, 1981] Chandra, A.K., Kozen, D.C., and Stockmeyer, L.J. 1981. Alternation, *Journal of the Assoc. Comput. Mach.*, 28:114-133.

[Chomsky, 1956] Chomsky, N. 1956. Three models for the description of language. *IRE Trans. on Information Theory*. 2(2):113-124.

[Chomsky and Miller, 1958] Chomsky, N. and Miller, G. 1958. Finite-state languages. *Information and Control*. 1:91-112.

[Cook, 1971] Cook, S. 1971. The complexity of theorem-proving procedures. *Proc. 3r ACM Symp. Theory of Comput.*, pp. 151-158.

[Cook and Reckhow, 1973] Cook, S. and Reckhow, R. 1973. Time bounded random access machines. *J. Comput. Syst. Sci.* 7:354-375.

[Davis, 1980] Davis, M. 1980. What is computation? in *Mathematics Today–Twelve Informal Essays*. L. Steen (ed.) 241-259.

[Floyd and Beigel, 1994] Floyd, R.W. and Beigel, R. 1994 *The Language of Machines: an Introduction to Computability and Formal Languages*. Computer Science Press, New York.

[Gurari, 1989] Gurari, E. 1989. *An Introduction to the Theory of Computation*. Computer Science Press, Rockville, MD.

[Harel, 1992] Harel, D. 1992. *Algorithmics: The spirit of Computing*. Addison-Wesley, Reading, MA.

[Hartmanis, 1994] Hartmanis, J. 1994. On computational complexity and the nature of computer science. *CACM*. 37(10):37-43.

[Hartmanis and Stearns, 1965] Hartmanis, J. and Stearns, R. 1965. On the computational complexity of algorithms. *Trans. Amer. Math. Soc.*. 117:285-306.

[Hopcroft and Ullman, 1979] Hopcroft, J. and Ullman, J. 1979. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, Reading, MA.

[Karp, 1972] Karp, R.M. 1972. Reducibility among combinatorial problems. in *Complexity of Computer Computations*, ed. R.E. Miller and J.W. Thatcher. Plenum Press, pp. 85-104.

[Karp and Ramachandran, 1990] Karp, R.M. and Ramachandran, V. 1990. Parallel algorithms for shared-memory machines. in *Handbook of Theoretical Computer Science*, ed. J. van Leeuwen, 869-941. Elsevier/MIT Press.

[Kleene, 1956] Kleene, S. 1956. Representation of events in nerve nets and finite automata. In *Automata Studies*, 3-41. Princeton University Press, NJ.

[Knuth *et al*, 1977] Knuth, D., Morris, J. and Pratt, V. 1977. Fast pattern matching in strings. *SIAM J. Comput.*. 6:323-350.

[Kohavi, 1978] Kohavi, Z. 1978. *Switching and Finite Automata Theory*. McGraw-Hill.

[Kolmogorov and Uspenskii, 1958] Kolmogorov, A. and Uspenskii, V. 1958. On the definition of an algorithm. *Uspekhi Mat. Nauk*. 13:3-28.

[Knuth, 1969] Knuth, D.E. 1969. *Fundamental Algorithms*, volume 1 of *The Art of Computer Programming*. Addison-Wesley, Reading, MA.

[Lesk, 1975] Lesk, M. 1975. LEX–a lexical analyzer generator. *Technical Report 39*. Bell Laboratories, Murray Hill, NJ.

[Levin, 1973] Levin, L. 1973. Universal sorting problems. *Problemi Peredachi Informatsii* 9(3):265-266. (In Russian)

[Lewis and Papadimitriou, 1981] Lewis, H. and Papadimitriou, C.H. 1981. *Elements of the Theory of Computation.* Prentice-Hall.

[Li and Vitányi, 1993] Li, M. and Vitányi, P. 1993. *An Introduction to Kolmogorov Complexity and Its Applications.* Springer-Verlag.

[McCulloch and Pitts, 1943] McCulloch, W. and Pitts, W. 1943. A logical calculus of ideas immanent in nervous activity. *Bull. Math. Biophysics*. 5:115-133.

[Lynch, 1996] Lynch, N. 1996. *Distributed Algorithms.* Morgan Kaufmann.

[Post, 1943] Post, E. 1943. Formal reductions of the general combinatorial decision problems. *Amer. J. Math.* 65:197-215.

[Rabin, 1963] Rabin, M.O. 1963. Real time computation. *Israel J. Math.* 1(4):203-211.

[Rabin and Scott, 1959] Rabin, M.O. and Scott, D. 1959. Finite automata and their decision problems. *IBM J. Res. Dev.*. 3:114-125.

[Robinson, 1991] Robinson, R. 1991. Minsky's small universal Turing machine. *International Journal of Mathematics*. 2(5):551-562.

[Ruzzo, 1981] Ruzzo, W.L. 1981. On uniform circuit complexity. *J. Comput. System Sci.* 22:365-383.

[Savitch, 1970] Savitch, J. 1970. Relationships between nondeterministic and deterministic tape complexities. *J. Comput. Syst. Sci.*. 4(2)177-192.

[Searls, 1993] Searls, D. 1993. The computational linguistics of biological sequences. In *Artificial Intelligence and Molecular Biology*. L. Hunter (ed.), MIT Press. 47-120.

[Schönhage, 1980] Schönhage, A. 1980. Storage modification machines. *SIAM J. Comput.*. 9:490-508.

[Schönhage and Strassen, 1971] Schönhage, A., and Strassen, V. 1971. Schnelle Multiplikation grosser Zahlen, *Computing* 7, 281-292.

[Sipser, 1997] Sipser, M. 1997 *Introduction to the Theory of Computation*, First Edition. PWS Publishing Company, Boston, MA.

[Turing, 1936] Turing, A. 1936. On computable numbers with an application to the Entscheidungsproblem. *Proc. London Math. Soc., series 2*, 42:230-265.

[van Emde Boas, 1990] van Emde Boas, P. 1990. Machine models and simulations. in *Handbook of Theoretical Computer Science*, ed. J. van Leeuwen, 1-66. Elsevier/MIT Press.

[Wood, 1987] Wood, D. 1987. *Theory of Computation*. Harper and Row.

[Yap, 1997] Yap, C. 1997. *Introduction to Complexity Classes*. Oxford University Press. To appear.

Further Information

The fundamentals of the theory of computation, automata theory, and formal languages can be found in Chapters 25–27 and in many text books including [Floyd and Beigel, 1994, Gurari, 1989, Harel, 1992, Hopcroft and Ullman, 1979, Lewis and Papadimitriou, 1981, Wood, 1987, Yap, 1997, Sipser, 1997]. One central focus of research in this area is to understand the relationships between different resource complexity classes. This work is motivated in part by some major open questions about the relationships between resources (such

as time and space) and the role of control mechanisms (such as nondeterminism or randomness). At the same time, new computational models are being introduced and studied. One recent model that has led to the resolution of a number of interesting problems is the *interactive proofs* (IP) model. IP is defined in terms of two Turing machines that communicate with each other. One of them has unlimited power and the other (called the *verifier*) is a probabilistic Turing machine whose time complexity is bounded by a polynomial. The study of IP has led to new ways to encrypt information as well as to the proof of some unexpected results about the difficulty of solving NP-hard problems (such as coloring, clique etc.) even approximately. See Chapter 35, sections 3 and 5. Another new model is the *quantum Turing machine*, which can solve in polynomial time some problems such as factoring that are believed to require exponential time on any hardware that follows the laws of "classical" (pre-quantum) physics. There are also attempts to use molecular or cell-level interactions as the basic operations of a computer.

The following annual conferences present the leading research work in computation theory: ACM Annual Symposium on Theory of Computing (STOC), IEEE Symposium on the Foundations of Computer Science (FOCS), IEEE Conference on Computational Complexity (CCC, formerly Structure in Complexity Theory), International Colloquium on Automata, Languages and Programming (ICALP), Symposium on Theoretical Aspects of Computer Science (STACS), Mathematical Foundations of Computer Science (MFCS), and Fundamentals of Computation Theory (FCT). There are many related conferences in the following areas: computational learning theory, computational geometry, algorithms, principles of distributed computing, computational biology, and database theory. In each case, specialized computational models and concrete algorithms are studied for a specific application area. There are also conferences in both pure and applied mathematics that admit topics in computation theory and complexity. We conclude with a partial list of major journals whose primary focus is in theory of computation: *Journal of the ACM*, *SIAM Journal on Computing*, *Journal of Computer and System Sciences*, *Information and Computation*, *Theory of Computing Systems* (formerly *Mathematical Systems Theory*), *Theoretical Computer Science*, *Computational Complexity*, *Journal of Complexity*, *Information Processing Letters*, *International Journal of Foundations of Computer Science*, and *Acta Informatica*.