Chapter 8

Introduction to Turing Machines

In this chapter we change our direction significantly. Until now, we have been interested primarily in simple classes of languages and the ways that they can be used for relatively constrained problems, such as analyzing protocols, searching text, or parsing programs. Now, we shall start looking at the question of what languages can be defined by any computational device whatsoever. This question is tantamount to the question of what computers can do, since recognizing the strings in a language is a formal way of expressing any problem, and solving a problem is a reasonable surrogate for what it is that computers do.

We begin with an informal argument, using an assumed knowledge of C programming, to show that there are specific problems we cannot solve using a computer. These problems are called "undecidable." We then introduce a venerable formalism for computers, called the Turing machine. While a Turing machine looks nothing like a PC, and would be grossly inefficient should some startup company decide to manufacture and sell them, the Turing machine long has been recognized as an accurate model for what any physical computing device is capable of doing.

In Chapter 9, we use the Turing machine to develop a theory of "undecidable" problems, that is, problems that no computer can solve. We show that a number of problems that are easy to express are in fact undecidable. An example is telling whether a given grammar is ambiguous, and we shall see many others.

8.1 Problems That Computers Cannot Solve

The purpose of this section is to provide an informal, C-programming-based introduction to the proof of a specific problem that computers cannot solve. The particular problem we discuss is whether the first thing a C program prints

is `hello, world`. Although we might imagine that simulation of the program would allow us to tell what the program does, we must in reality contend with programs that take an unimaginably long time before making any output at all. This problem — not knowing when, if ever, something will occur — is the ultimate cause of our inability to tell what a program does. However, proving formally that there is no program to do a stated task is quite tricky, and we need to develop some formal mechanics. In this section, we give the intuition behind the formal proofs.

8.1.1 Programs that Print "Hello, World"

In Fig. 8.1 is the first C program met by students who read Kernighan and Ritchie's classic book.[1] It is rather easy to discover that this program prints `hello, world` and terminates. This program is so transparent that it has become a common practice to introduce languages by showing how to write a program to print `hello, world` in those languages.

```
main()
{
    printf("hello, world\n");
}
```

Figure 8.1: Kernighan and Ritchie's hello-world program

However, there are other programs that also print `hello, world`; yet the fact that they do so is far from obvious. Figure 8.2 shows another program that might print `hello, world`. It takes an input n, and looks for positive integer solutions to the equation $x^n + y^n = z^n$. If it finds one, it prints `hello, world`. If it never finds integers x, y, and z to satisfy the equation, then it continues searching forever, and never prints `hello, world`.

To understand what this program does, first observe that `exp` is an auxiliary function to compute exponentials. The main program needs to search through triples (x, y, z) in an order such that we are sure we get to every triple of positive integers eventually. To organize the search properly, we use a fourth variable, `total`, that starts at 3 and, in the while-loop, is increased one unit at a time, eventually reaching any finite integer. Inside the while-loop, we divide `total` into three positive integers x, y, and z, by first allowing x to range from 1 to `total-2`, and within that for-loop allowing y to range from 1 up to one less than what x has not already taken from `total`. What remains, which must be between 1 and `total-2`, is given to z.

In the innermost loop, the triple (x, y, z) is tested to see if $x^n + y^n = z^n$. If so, the program prints `hello, world`, and if not, it prints nothing.

[1] B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, 1978, Prentice-Hall, Englewood Cliffs, NJ.

```
int exp(int i, n)
/* computes i to the power n */
{
    int ans, j;
    ans = 1;
    for (j=1; j<=n; j++) ans *= i;
    return(ans);
}

main ()
{
    int n, total, x, y, z;
    scanf("%d", &n);
    total = 3;
    while (1) {
        for (x=1; x<=total-2; x++)
            for (y=1; y<=total-x-1; y++) {
                z = total - x - y;
                if (exp(x,n) + exp(y,n) == exp(z,n))
                    printf("hello, world\n");
            }
        total++;
    }
}
```

Figure 8.2: Fermat's last theorem expressed as a hello-world program

If the value of n that the program reads is 2, then it will eventually find combinations of integers such as `total` $= 12$, $x = 3$, $y = 4$, and $z = 5$, for which $x^n + y^n = z^n$. Thus, for input 2, the program *does* print `hello, world`.

However, for any integer $n > 2$, the program will never find a triple of positive integers to satisfy $x^n + y^n = z^n$, and thus will fail to print `hello, world`. Interestingly, until a few years ago, it was not known whether or not this program would print `hello, world` for some large integer n. The claim that it would not, i.e., that there are no integer solutions to the equation $x^n + y^n = z^n$ if $n > 2$, was made by Fermat 300 years ago, but no proof was found until quite recently. This statement is often referred to as "Fermat's last theorem."

Let us define the *hello-world problem* to be: determine whether a given C program, with a given input, prints `hello, world` as the first 12 characters that it prints. In what follows, we often use, as a shorthand, the statement about a program that it prints `hello, world` to mean that it prints `hello, world` as the first 12 characters that it prints.

It seems likely that, if it takes mathematicians 300 years to resolve a question about a single, 22-line program, then the general problem of telling whether a

Why Undecidable Problems Must Exist

While it is tricky to prove that a specific problem, such as the "hello-world problem" discussed here, must be undecidable, it is quite easy to see why almost all problems must be undecidable by any system that involves programming. Recall that a "problem" is really membership of a string in a language. The number of different languages over any alphabet of more than one symbol is not countable. That is, there is no way to assign integers to the languages such that every language has an integer, and every integer is assigned to one language.

On the other hand programs, being finite strings over a finite alphabet (typically a subset of the ASCII alphabet), *are* countable. That is, we can order them by length, and for programs of the same length, order them lexicographically. Thus, we can speak of the first program, the second program, and in general, the ith program for any integer i.

As a result, we know there are infinitely fewer programs than there are problems. If we picked a language at random, almost certainly it would be an undecidable problem. The only reason that most problems *appear* to be decidable is that we rarely are interested in random problems. Rather, we tend to look at fairly simple, well-structured problems, and indeed these are often decidable. However, even among the problems we are interested in and can state clearly and succinctly, we find many that are undecidable; the hello-world problem is a case in point.

given program, on a given input, prints `hello, world` must be hard indeed. In fact, any of the problems that mathematicians have not yet been able to resolve can be turned into a question of the form "does this program, with this input, print `hello, world`?" Thus, it would be remarkable indeed if we could write a program that could examine any program P and input I for P, and tell whether P, run with I as its input, would print `hello, world`. We shall prove that no such program exists.

8.1.2 The Hypothetical "Hello, World" Tester

The proof of impossibility of making the hello-world test is a proof by contradiction. That is, we assume there is a program, call it H, that takes as input a program P and an input I, and tells whether P with input I prints `hello, world`. Figure 8.3 is a representation of what H does. In particular, the only output H makes is either to print the three characters `yes` or to print the two characters `no`. It always does one or the other.

If a problem has an algorithm like H, that always tells correctly whether an instance of the problem has answer "yes" or "no," then the problem is said to be "decidable." Otherwise, the problem is "undecidable." Our goal is to prove

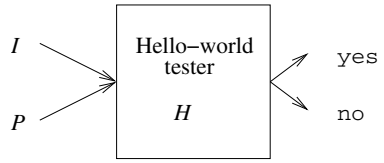

Figure 8.3: A hypothetical program H that is a hello-world detector

that H doesn't exist; i.e., the hello-world problem is undecidable.

In order to prove that statement by contradiction, we are going to make several changes to H, eventually constructing a related program called H_2 that we show does not exist. Since the changes to H are simple transformations that can be done to any C program, the only questionable statement is the existence of H, so it is that assumption we have contradicted.

To simplify our discussion, we shall make a few assumptions about C programs. These assumptions make H's job easier, not harder, so if we can show a "hello-world tester" for these restricted programs does not exist, then surely there is no such tester that could work for a broader class of programs. Our assumptions are:

1. All output is character-based, e.g., we are not using a graphics package or any other facility to make output that is not in the form of characters.

2. All character-based output is performed using `printf`, rather than `putchar()` or another character-based output function.

We now assume that the program H exists. Our first modification is to change the output `no`, which is the response that H makes when its input program P does not print `hello, world` as its first output in response to input I. As soon as H prints "n," we know it will eventually follow with the "o."[2] Thus, we can modify any `printf` statement in H that prints "n" to instead print `hello, world`. Another `printf` statement that prints an "o" but not the "n" is omitted. As a result, the new program, which we call H_1, behaves like H, except it prints `hello, world` exactly when H would print `no`. H_1 is suggested by Fig. 8.4.

Our next transformation on the program is a bit trickier; it is essentially the insight that allowed Alan Turing to prove his undecidability result about Turing machines. Since we are really interested in programs that take other programs as input and tell something about them, we shall restrict H_1 so it:

a) Takes only input P, not P and I.

b) Asks what P would do if its input were its own code, i.e., what would H_1 do on inputs P as program and P as input I as well?

[2] Most likely, the program would put `no` in one `printf`, but it could print the "n" in one `printf` and the "o" in another.

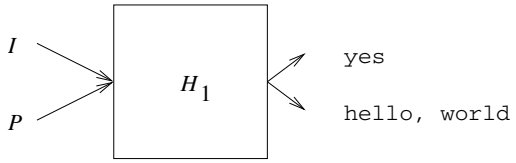

Figure 8.4: H_1 behaves like H, but it says `hello, world` instead of `no`

The modifications we must perform on H_1 to produce the program H_2 suggested in Fig. 8.5 are as follows:

1. H_2 first reads the entire input P and stores it in an array A, which it "malloc's" for the purpose.[3]

2. H_2 then simulates H_1, but whenever H_1 would read input from P or I, H_2 reads from the stored copy in A. To keep track of how much of P and I H_1 has read, H_2 can maintain two cursors that mark positions in A.

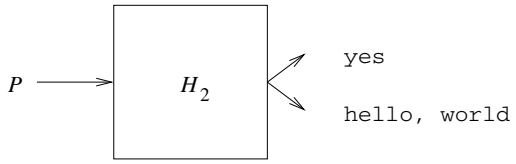

Figure 8.5: H_2 behaves like H_1, but uses its input P as both P and I

We are now ready to prove H_2 cannot exist. Thus, H_1 does not exist, and likewise, H does not exist. The heart of the argument is to envision what H_2 does when given itself as input. This situation is suggested in Fig. 8.6. Recall that H_2, given any program P as input, makes output `yes` if P prints `hello, world` when given itself as input. Also, H_2 prints `hello, world` if P, given itself as input, does not print `hello, world` as its first output.

Suppose that the H_2 represented by the box in Fig. 8.6 makes the output `yes`. Then the H_2 in the box is saying about its input H_2 that H_2, given itself as input, prints `hello, world` as its first output. But we just supposed that the first output H_2 makes in this situation is `yes` rather than `hello, world`.

Thus, it appears that in Fig. 8.6 the output of the box is `hello, world`, since it must be one or the other. But if H_2, given itself as input, prints `hello, world` first, then the output of the box in Fig. 8.6 must be `yes`. Whichever output we suppose H_2 makes, we can argue that it makes the other output.

[3] The UNIX `malloc` system function allocates a block of memory of a size specified in the call to `malloc`. This function is used when the amount of storage needed cannot be determined until the program is run, as would be the case if an input of arbitrary length were read. Typically, `malloc` would be called several times, as more and more input is read and progressively more space is needed.

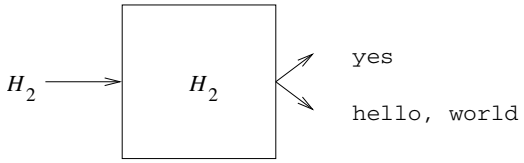

Figure 8.6: What does H_2 do when given itself as input?

This situation is paradoxical, and we conclude that H_2 cannot exist. As a result, we have contradicted the assumption that H exists. That is, we have proved that no program H can tell whether or not a given program P with input I prints `hello, world` as its first output.

8.1.3 Reducing One Problem to Another

Now, we have one problem — does a given program with given input print `hello, world` as the first thing it prints? — that we know no computer program can solve. A problem that cannot be solved by computer is called *undecidable*. We shall give the formal definition of "undecidable" in Section 9.3, but for the moment, let us use the term informally. Suppose we want to determine whether or not some other problem is solvable by a computer. We can try to write a program to solve it, but if we cannot figure out how to do so, then we might try a proof that there is no such program.

Perhaps we could prove this new problem undecidable by a technique similar to what we did for the hello-world problem: assume there is a program to solve it and develop a paradoxical program that must do two contradictory things, like the program H_2. However, once we have one problem that we know is undecidable, we no longer have to prove the existence of a paradoxical situation. It is sufficient to show that if we could solve the new problem, then we could use that solution to solve a problem we already know is undecidable. The strategy is suggested in Fig. 8.7; the technique is called the *reduction* of P_1 to P_2.

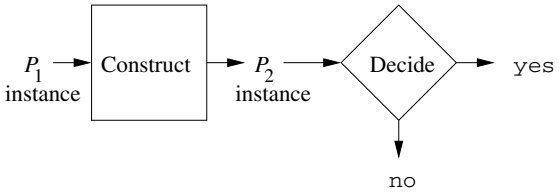

Figure 8.7: If we could solve problem P_2, then we could use its solution to solve problem P_1

Suppose that we know problem P_1 is undecidable, and P_2 is a new problem that we would like to prove is undecidable as well. We suppose that there is a program represented in Fig. 8.7 by the diamond labeled "decide"; this program

Can a Computer Really Do All That?

If we examine a program such as Fig. 8.2, we might ask whether it really searches for counterexamples to Fermat's last theorem. After all, integers are only 32 bits long in the typical computer, and if the smallest counterexample involved integers in the billions, there would be an overflow error before the solution was found. In fact, one could argue that a computer with 128 megabytes of main memory and a 30 gigabyte disk, has "only" $256^{30128000000}$ states, and is thus a finite automaton.

However, treating computers as finite automata (or treating brains as finite automata, which is where the FA idea originated), is unproductive. The number of states involved is so large, and the limits so unclear, that you don't draw any useful conclusions. In fact, there is every reason to believe that, if we wanted to, we could expand the set of states of a computer arbitrarily.

For instance, we can represent integers as linked lists of digits, of arbitrary length. If we run out of memory, the program can print a request for a human to dismount its disk, store it, and replace it by an empty disk. As time goes on, the computer could print requests to swap among as many disks as the computer needs. This program would be far more complex than that of Fig. 8.2, but not beyond our capabilities to write. Similar tricks would allow any other program to avoid finite limitations on the size of memory or on the size of integers or other data items.

prints `yes` or `no`, depending on whether its input instance of problem P_2 is or is not in the language of that problem.[4]

In order to make a proof that problem P_2 is undecidable, we have to invent a construction, represented by the square box in Fig. 8.7, that converts instances of P_1 to instances of P_2 that have the same answer. That is, any string in the language P_1 is converted to some string in the language P_2, and any string over the alphabet of P_1 that is *not* in the language P_1 is converted to a string that is not in the language P_2. Once we have this construction, we can solve P_1 as follows:

1. Given an instance of P_1, that is, given a string w that may or may not be in the language P_1, apply the construction algorithm to produce a string x.

2. Test whether x is in P_2, and give the same answer about w and P_1.

[4] Recall that a problem is really a language. When we talked of the problem of deciding whether a given program and input results in `hello, world` as the first output, we were really talking about strings consisting of a C source program followed by whatever input file(s) the program reads. This set of strings is a language over the alphabet of ASCII characters.

The Direction of a Reduction Is Important

It is a common mistake to try to prove a problem P_2 undecidable by reducing P_2 to some known undecidable problem P_1; i.e., showing the statement "if P_1 is decidable, then P_2 is decidable." That statement, although surely true, is useless, since its hypothesis "P_1 is decidable" is false.

The only way to prove a new problem P_2 to be undecidable is to reduce a known undecidable problem P_1 to P_2. That way, we prove the statement "if P_2 is decidable, then P_1 is decidable." The contrapositive of that statement is "if P_1 is undecidable, then P_2 is undecidable." Since we know that P_1 undecidable, we can deduce that P_2 is undecidable.

If w is in P_1, then x is in P_2, so this algorithm says `yes`. If w is not in P_1, then x is not in P_2, and the algorithm says `no`. Either way, it says the truth about w. Since we assumed that no algorithm to decide membership of a string in P_1 exists, we have a proof by contradiction that the hypothesized decision algorithm for P_2 does not exist; i.e., P_2 is undecidable.

Example 8.1: Let us use this methodology to show that the question "does program Q, given input y, ever call function `foo`" is undecidable. Note that Q may not have a function `foo`, in which case the problem is easy, but the hard cases are when Q has a function `foo` but may or may not reach a call to `foo` with input y. Since we only know one undecidable problem, the role of P_1 in Fig. 8.7 will be played by the hello-world problem. P_2 will be the *calls-foo problem* just mentioned. We suppose there is a program that solves the calls-foo problem. Our job is to design an algorithm that converts the hello-world problem into the calls-foo problem.

That is, given program Q and its input y, we must construct a program R and an input z such that R, with input z, calls `foo` if and only if Q with input y prints `hello, world`. The construction is not hard:

1. If Q has a function called `foo`, rename it and all calls to that function. Clearly the new program Q_1 does exactly what Q does.

2. Add to Q_1 a function `foo`. This function does nothing, and is not called. The resulting program is Q_2.

3. Modify Q_2 to remember the first 12 characters that it prints, storing them in a global array A. Let the resulting program be Q_3.

4. Modify Q_3 so that whenever it executes any output statement, it then checks in the array A to see if it has written 12 characters or more, and if so, whether `hello, world` are the first 12 characters. In that case, call

the new function `foo` that was added in item (2). The resulting program is R, and input z is the same as y.

Suppose that Q with input y prints `hello, world` as its first output. Then R as constructed will call `foo`. However, if Q with input y does not print `hello, world` as its first output, then R will never call `foo`. If we can decide whether R with input z calls `foo`, then we also know whether Q with input y (remember $y = z$) prints `hello, world`. Since we know that no algorithm to decide the hello-world problem exists, and all four steps of the construction of R from Q could be carried out by a program that edited the code of programs, our assumption that there was a calls-foo tester is wrong. No such program exists, and the calls-foo problem is undecidable. □

8.1.4 Exercises for Section 8.1

Exercise 8.1.1: Give reductions from the hello-world problem to each of the problems below. Use the informal style of this section for describing plausible program transformations, and do not worry about the real limits such as maximum file size or memory size that real computers impose.

***!** a) Given a program and an input, does the program eventually halt; i.e., does the program not loop forever on the input?

b) Given a program and an input, does the program ever produce *any* output?

! c) Given two programs and an input, do the programs produce the same output for the given input?

8.2 The Turing Machine

The purpose of the theory of undecidable problems is not only to establish the existence of such problems — an intellectually exciting idea in its own right — but to provide guidance to programmers about what they might or might not be able to accomplish through programming. The theory also has great pragmatic impact when we discuss, as we shall in Chapter 10, problems that although decidable, require large amounts of time to solve them. These problems, called "intractable problems," tend to present greater difficulty to the programmer and system designer than do the undecidable problems. The reason is that, while undecidable problems are usually quite obviously so, and their solutions are rarely attempted in practice, the intractable problems are faced every day. Moreover, they often yield to small modifications in the requirements or to heuristic solutions. Thus, the designer is faced quite frequently with having to decide whether or not a problem is in the intractable class, and what to do about it, if so.

We need tools that will allow us to prove everyday questions undecidable or intractable. The technology introduced in Section 8.1 is useful for questions that deal with programs, but it does not translate easily to problems in unrelated domains. For example, we would have great difficulty reducing the hello-world problem to the question of whether a grammar is ambiguous.

As a result, we need to rebuild our theory of undecidability, based not on programs in C or another language, but based on a very simple model of a computer, called the Turing machine. This device is essentially a finite automaton that has a single tape of infinite length on which it may read and write data. One advantage of the Turing machine over programs as representation of what can be computed is that the Turing machine is sufficiently simple that we can represent its configuration precisely, using a simple notation much like the ID's of a PDA. In comparison, while C programs have a state, involving all the variables in whatever sequence of function calls have been made, the notation for describing these states is far too complex to allow us to make understandable, formal proofs.

Using the Turing machine notation, we shall prove undecidable certain problems that appear unrelated to programming. For instance, we shall show in Section 9.4 that "Post's Correspondence Problem," a simple question involving two lists of strings, is undecidable, and this problem makes it easy to show questions about grammars, such as ambiguity, to be undecidable. Likewise, when we introduce intractable problems we shall find that certain questions, seemingly having little to do with computation (e.g., satisfiability of boolean formulas), are intractable.

8.2.1 The Quest to Decide All Mathematical Questions

At the turn of the 20th century, the mathematician D. Hilbert asked whether it was possible to find an algorithm for determining the truth or falsehood of any mathematical proposition. In particular, he asked if there was a way to determine whether any formula in the first-order predicate calculus, applied to integers, was true. Since the first-order predicate calculus of integers is sufficiently powerful to express statements like "this grammar is ambiguous," or "this program prints `hello, world`," had Hilbert been successful, these problems would have algorithms that we now know do not exist.

However, in 1931, K. Gödel published his famous incompleteness theorem. He constructed a formula in the predicate calculus applied to integers, which asserted that the formula itself could be neither proved nor disproved within the predicate calculus. Gödel's technique resembles the construction of the self-contradictory program H_2 in Section 8.1.2, but deals with functions on the integers, rather than with C programs.

The predicate calculus was not the only notion that mathematicians had for "any possible computation." In fact predicate calculus, being declarative rather than computational, had to compete with a variety of notations, including the "partial-recursive functions," a rather programming-language-like notation, and

other similar notations. In 1936, A. M. Turing proposed the Turing machine as a model of "any possible computation." This model is computer-like, rather than program-like, even though true electronic, or even electromechanical computers were several years in the future (and Turing himself was involved in the construction of such a machine during World War II).

Interestingly, all the serious proposals for a model of computation have the same power; that is, they compute the same functions or recognize the same languages. The unprovable assumption that any general way to compute will allow us to compute only the partial-recursive functions (or equivalently, what Turing machines or modern-day computers can compute) is known as *Church's hypothesis* (after the logician A. Church) or the *Church-Turing thesis*.

8.2.2 Notation for the Turing Machine

We may visualize a Turing machine as in Fig. 8.8. The machine consists of a *finite control*, which can be in any of a finite set of states. There is a *tape* divided into squares or *cells*; each cell can hold any one of a finite number of symbols.

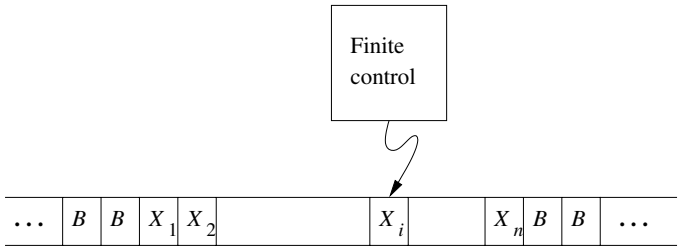

Figure 8.8: A Turing machine

Initially, the *input*, which is a finite-length string of symbols chosen from the *input alphabet*, is placed on the tape. All other tape cells, extending infinitely to the left and right, initially hold a special symbol called the *blank*. The blank is a *tape symbol*, but not an input symbol, and there may be other tape symbols besides the input symbols and the blank, as well.

There is a *tape head* that is always positioned at one of the tape cells. The Turing machine is said to be *scanning* that cell. Initially, the tape head is at the leftmost cell that holds the input.

A *move* of the Turing machine is a function of the state of the finite control and the tape symbol scanned. In one move, the Turing machine will:

1. Change state. The next state optionally may be the same as the current state.

2. Write a tape symbol in the cell scanned. This tape symbol replaces whatever symbol was in that cell. Optionally, the symbol written may be the same as the symbol currently there.

3. Move the tape head left or right. In our formalism we require a move, and do not allow the head to remain stationary. This restriction does not constrain what a Turing machine can compute, since any sequence of moves with a stationary head could be condensed, along with the next tape-head move, into a single state change, a new tape symbol, and a move left or right.

The formal notation we shall use for a *Turing machine* (TM) is similar to that used for finite automata or PDA's. We describe a TM by the 7-tuple

$$M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$$

whose components have the following meanings:

Q: The finite set of *states* of the finite control.

Σ: The finite set of *input symbols*.

Γ: The complete set of *tape symbols*; Σ is always a subset of Γ.

δ: The *transition function*. The arguments of $\delta(q, X)$ are a state q and a tape symbol X. The value of $\delta(q, X)$, if it is defined, is a triple (p, Y, D), where:

1. p is the next state, in Q.
2. Y is the symbol, in Γ, written in the cell being scanned, replacing whatever symbol was there.
3. D is a *direction*, either L or R, standing for "left" or "right," respectively, and telling us the direction in which the head moves.

q_0: The *start state*, a member of Q, in which the finite control is found initially.

B: The *blank* symbol. This symbol is in Γ but not in Σ; i.e., it is not an input symbol. The blank appears initially in all but the finite number of initial cells that hold input symbols.

F: The set of *final* or *accepting* states, a subset of Q.

8.2.3 Instantaneous Descriptions for Turing Machines

In order to describe formally what a Turing machine does, we need to develop a notation for configurations or *instantaneous descriptions* (ID's), like the notation we developed for PDA's. Since a TM, in principle, has an infinitely long tape, we might imagine that it is impossible to describe the configurations of a TM succinctly. However, after any finite number of moves, the TM can have visited only a finite number of cells, even though the number of cells visited can eventually grow beyond any finite limit. Thus, in every ID, there is an infinite prefix and an infinite suffix of cells that have never been visited. These cells

must all hold either blanks or one of the finite number of input symbols. We thus show in an ID only the cells between the leftmost and the rightmost nonblanks. Under special conditions, when the head is scanning one of the leading or trailing blanks, a finite number of blanks to the left or right of the nonblank portion of the tape must also be included in the ID.

In addition to representing the tape, we must represent the finite control and the tape-head position. To do so, we embed the state in the tape, and place it immediately to the left of the cell scanned. To disambiguate the tape-plus-state string, we have to make sure that we do not use as a state any symbol that is also a tape symbol. However, it is easy to change the names of the states so they have nothing in common with the tape symbols, since the operation of the TM does not depend on what the states are called. Thus, we shall use the string $X_1X_2\cdots X_{i-1}qX_iX_{i+1}\cdots X_n$ to represent an ID in which

1. q is the state of the Turing machine.
2. The tape head is scanning the ith symbol from the left.
3. $X_1X_2\cdots X_n$ is the portion of the tape between the leftmost and the rightmost nonblank. As an exception, if the head is to the left of the leftmost nonblank or to the right of the rightmost nonblank, then some prefix or suffix of $X_1X_2\cdots X_n$ will be blank, and i will be 1 or n, respectively.

We describe moves of a Turing machine $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$ by the $\vdash_M$ notation that was used for PDA's. When the TM M is understood, we shall use just $\vdash$ to reflect moves. As usual, $\vdash_M^*$, or just $\vdash^*$, will be used to indicate zero, one, or more moves of the TM M.

Suppose $\delta(q, X_i) = (p, Y, L)$; i.e., the next move is leftward. Then

$$X_1X_2\cdots X_{i-1}qX_iX_{i+1}\cdots X_n \vdash_M X_1X_2\cdots X_{i-2}pX_{i-1}YX_{i+1}\cdots X_n$$

Notice how this move reflects the change to state p and the fact that the tape head is now positioned at cell $i-1$. There are two important exceptions:

1. If $i = 1$, then M moves to the blank to the left of X_1. In that case,
$$qX_1X_2\cdots X_n \vdash_M pBYX_2\cdots X_n$$
2. If $i = n$ and $Y = B$, then the symbol B written over X_n joins the infinite sequence of trailing blanks and does not appear in the next ID. Thus,
$$X_1X_2\cdots X_{n-1}qX_n \vdash_M X_1X_2\cdots X_{n-2}pX_n$$

Now, suppose $\delta(q, X_i) = (p, Y, R)$; i.e., the next move is rightward. Then

$$X_1X_2\cdots X_{i-1}qX_iX_{i+1}\cdots X_n \vdash_M X_1X_2\cdots X_{i-1}YpX_{i+1}\cdots X_n$$

Here, the move reflects the fact that the head has moved to cell $i+1$. Again there are two important exceptions:

1. If $i = n$, then the $i+1$st cell holds a blank, and that cell was not part of the previous ID. Thus, we instead have

$$X_1X_2\cdots X_{n-1}qX_n \underset{M}{\vdash} X_1X_2\cdots X_{n-1}YpB$$

2. If $i = 1$ and $Y = B$, then the symbol B written over X_1 joins the infinite sequence of leading blanks and does not appear in the next ID. Thus,

$$qX_1X_2\cdots X_n \underset{M}{\vdash} pX_2\cdots X_n$$

Example 8.2: Let us design a Turing machine and see how it behaves on a typical input. The TM we construct will accept the language $\{0^n1^n \mid n \geq 1\}$. Initially, it is given a finite sequence of 0's and 1's on its tape, preceded and followed by an infinity of blanks. Alternately, the TM will change a 0 to an X and then a 1 to a Y, until all 0's and 1's have been matched.

In more detail, starting at the left end of the input, it enters a loop in which it changes a 0 to an X and moves to the right over whatever 0's and Y's it sees, until it comes to a 1. It changes the 1 to a Y, and moves left, over Y's and 0's, until it finds an X. At that point, it looks for a 0 immediately to the right, and if it finds one, changes it to X and repeats the process, changing a matching 1 to a Y.

If the nonblank input is not in $\mathbf{0^*1^*}$, then the TM will eventually fail to have a next move and will die without accepting. However, if it finishes changing all the 0's to X's on the same round it changes the last 1 to a Y, then it has found its input to be of the form 0^n1^n and accepts. The formal specification of the TM M is

$$M = (\{q_0, q_1, q_2, q_3, q_4\}, \{0,1\}, \{0,1,X,Y,B\}, \delta, q_0, B, \{q_4\})$$

where δ is given by the table in Fig. 8.9.

	Symbol				
State	0	1	X	Y	B
q_0	(q_1, X, R)	–	–	(q_3, Y, R)	–
q_1	$(q_1, 0, R)$	(q_2, Y, L)	–	(q_1, Y, R)	–
q_2	$(q_2, 0, L)$	–	(q_0, X, R)	(q_2, Y, L)	–
q_3	–	–	–	(q_3, Y, R)	(q_4, B, R)
q_4	–	–	–	–	–

Figure 8.9: A Turing machine to accept $\{0^n1^n \mid n \geq 1\}$

As M performs its computation, the portion of the tape, where M's tape head has visited, will always be a sequence of symbols described by the regular expression $\mathbf{X^*0^*Y^*1^*}$. That is, there will be some 0's that have been changed to X's, followed by some 0's that have not yet been changed to X's. Then there

are some 1's that were changed to Y's, and 1's that have not yet been changed to Y's. There may or may not be some 0's and 1's following.

State q_0 is the initial state, and M also enters state q_0 every time it returns to the leftmost remaining 0. If M is in state q_0 and scanning a 0, the rule in the upper-left corner of Fig. 8.9 tells it to go to state q_1, change the 0 to an X, and move right. Once in state q_1, M keeps moving right over all 0's and Y's that it finds on the tape, remaining in state q_1. If M sees an X or a B, it dies. However, if M sees a 1 when in state q_1, it changes that 1 to a Y, enters state q_2, and starts moving left.

In state q_2, M moves left over 0's and Y's, remaining in state q_2. When M reaches the rightmost X, which marks the right end of the block of 0's that have already been changed to X, M returns to state q_0 and moves right. There are two cases:

1. If M now sees a 0, then it repeats the matching cycle we have just described.

2. If M sees a Y, then it has changed all the 0's to X's. If all the 1's have been changed to Y's, then the input was of the form 0^n1^n, and M should accept. Thus, M enters state q_3, and starts moving right, over Y's. If the first symbol other than a Y that M sees is a blank, then indeed there were an equal number of 0's and 1's, so M enters state q_4 and accepts. On the other hand, if M encounters another 1, then there are too many 1's, so M dies without accepting. If it encounters a 0, then the input was of the wrong form, and M also dies.

Here is an example of an accepting computation by M. Its input is 0011. Initially, M is in state q_0, scanning the first 0, i.e., M's initial ID is q_00011. The entire sequence of moves of M is:

$$q_00011 \vdash Xq_1011 \vdash X0q_111 \vdash Xq_20Y1 \vdash q_2X0Y1 \vdash$$
$$Xq_00Y1 \vdash XXq_1Y1 \vdash XXYq_11 \vdash XXq_2YY \vdash Xq_2XYY \vdash$$
$$XXq_0YY \vdash XXYq_3Y \vdash XXYYq_3B \vdash XXYYBq_4B$$

For another example, consider what M does on the input 0010, which is not in the language accepted.

$$q_00010 \vdash Xq_1010 \vdash X0q_110 \vdash Xq_20Y0 \vdash q_2X0Y0 \vdash$$
$$Xq_00Y0 \vdash XXq_1Y0 \vdash XXYq_10 \vdash XXY0q_1B$$

The behavior of M on 0010 resembles the behavior on 0011, until in ID $XXYq_10$ M scans the final 0 for the first time. M must move right, staying in state q_1, which takes it to the ID $XXY0q_1B$. However, in state q_1 M has no move on tape symbol B; thus M dies and does not accept its input. $\Box$

8.2.4 Transition Diagrams for Turing Machines

We can represent the transitions of a Turing machine pictorially, much as we did for the PDA. A *transition diagram* consists of a set of nodes corresponding to the states of the TM. An arc from state q to state p is labeled by one or more items of the form X/YD, where X and Y are tape symbols, and D is a direction, either L or R. That is, whenever $\delta(q, X) = (p, Y, D)$, we find the label X/YD on the arc from q to p. However, in our diagrams, the direction D is represented pictorially by $\leftarrow$ for "left" and $\rightarrow$ for "right."

As for other kinds of transition diagrams, we represent the start state by the word "Start" and an arrow entering that state. Accepting states are indicated by double circles. Thus, the only information about the TM one cannot read directly from the diagram is the symbol used for the blank. We shall assume that symbol is B unless we state otherwise.

Example 8.3: Figure 8.10 shows the transition diagram for the Turing machine of Example 8.2, whose transition function was given in Fig. 8.9. □

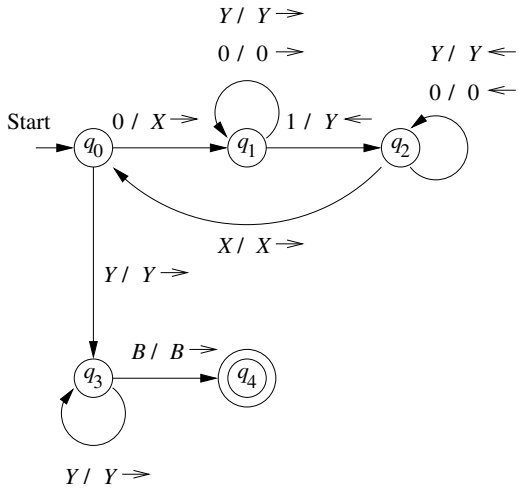

Figure 8.10: Transition diagram for a TM that accepts strings of the form 0^n1^n

Example 8.4: While today we find it most convenient to think of Turing machines as recognizers of languages, or equivalently, solvers of problems, Turing's original view of his machine was as a computer of integer-valued functions. In his scheme, integers were represented in unary, as blocks of a single character, and the machine computed by changing the lengths of the blocks or by constructing new blocks elsewhere on the tape. In this simple example, we shall show how a Turing machine might compute the function $\dot{-}$, which is called *monus* or *proper subtraction* and is defined by $m \dot{-} n = \max(m - n, 0)$. That is, $m \dot{-} n$ is $m - n$ if $m \geq n$ and 0 if $m < n$.

A TM that performs this operation is specified by

$$M = (\{q_0, q_1, \ldots, q_6\}, \{0,1\}, \{0,1,B\}, \delta, q_0, B)$$

Note that, since this TM is not used to accept inputs, we have omitted the seventh component, which is the set of accepting states. M will start with a tape consisting of 0^m10^n surrounded by blanks. M halts with $0^{m \dot{-} n}$ on its tape, surrounded by blanks.

M repeatedly finds its leftmost remaining 0 and replaces it by a blank. It then searches right, looking for a 1. After finding a 1, it continues right, until it comes to a 0, which it replaces by a 1. M then returns left, seeking the leftmost 0, which it identifies when it first meets a blank and then moves one cell to the right. The repetition ends if either:

1. Searching right for a 0, M encounters a blank. Then the n 0's in 0^m10^n have all been changed to 1's, and $n+1$ of the m 0's have been changed to B. M replaces the $n+1$ 1's by one 0 and n B's, leaving $m-n$ 0's on the tape. Since $m \geq n$ in this case, $m - n = m \dot{-} n$.

2. Beginning the cycle, M cannot find a 0 to change to a blank, because the first m 0's already have been changed to B. Then $n \geq m$, so $m \dot{-} n = 0$. M replaces all remaining 1's and 0's by B and ends with a completely blank tape.

Figure 8.11 gives the rules of the transition function δ, and we have also represented δ as a transition diagram in Fig. 8.12. The following is a summary of the role played by each of the seven states:

q_0: This state begins the cycle, and also breaks the cycle when appropriate. If M is scanning a 0, the cycle must repeat. The 0 is replaced by B, the head moves right, and state q_1 is entered. On the other hand, if M is scanning 1, then all possible matches between the two groups of 0's on the tape have been made, and M goes to state q_5 to make the tape blank.

q_1: In this state, M searches right, through the initial block of 0's, looking for the leftmost 1. When found, M goes to state q_2.

q_2: M moves right, skipping over 1's, until it finds a 0. It changes that 0 to a 1, turns leftward, and enters state q_3. However, it is also possible that there are no more 0's left after the block of 1's. In that case, M in state q_2 encounters a blank. We have case (1) described above, where n 0's in the second block of 0's have been used to cancel n of the m 0's in the first block, and the subtraction is complete. M enters state q_4, whose purpose is to convert the 1's on the tape to blanks.

q_3: M moves left, skipping over 0's and 1's, until it finds a blank. When it finds B, it moves right and returns to state q_0, beginning the cycle again.

State	Symbol 0	1	B
q_0	(q_1, B, R)	(q_5, B, R)	–
q_1	$(q_1, 0, R)$	$(q_2, 1, R)$	–
q_2	$(q_3, 1, L)$	$(q_2, 1, R)$	(q_4, B, L)
q_3	$(q_3, 0, L)$	$(q_3, 1, L)$	(q_0, B, R)
q_4	$(q_4, 0, L)$	(q_4, B, L)	$(q_6, 0, R)$
q_5	(q_5, B, R)	(q_5, B, R)	(q_6, B, R)
q_6	–	–	–

Figure 8.11: A Turing machine that computes the proper-subtraction function

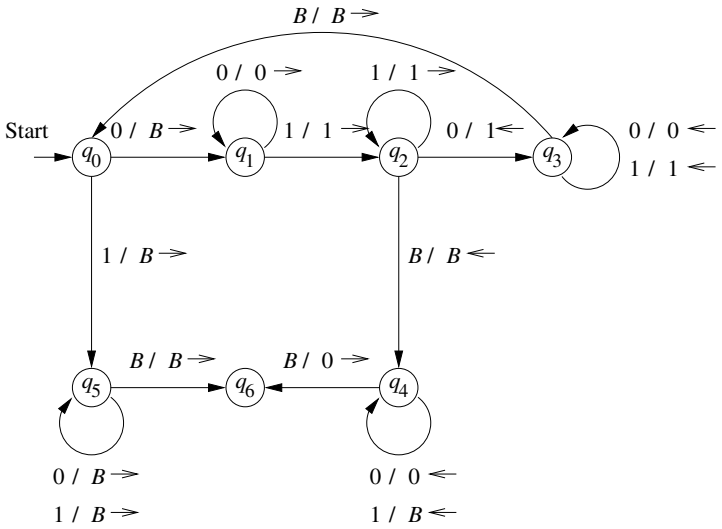

Figure 8.12: Transition diagram for the TM of Example 8.4

q_4: Here, the subtraction is complete, but one unmatched 0 in the first block was incorrectly changed to a B. M therefore moves left, changing 1's to B's, until it encounters a B on the tape. It changes that B back to 0, and enters state q_6, wherein M halts.

q_5: State q_5 is entered from q_0 when it is found that all 0's in the first block have been changed to B. In this case, described in (2) above, the result of the proper subtraction is 0. M changes all remaining 0's and 1's to B and enters state q_6.

q_6: The sole purpose of this state is to allow M to halt when it has finished its task. If the subtraction had been a subroutine of some more complex function, then q_6 would initiate the next step of that larger computation.

□

8.2.5 The Language of a Turing Machine

We have intuitively suggested the way that a Turing machine accepts a language. The input string is placed on the tape, and the tape head begins at the leftmost input symbol. If the TM eventually enters an accepting state, then the input is accepted, and otherwise not.

More formally, let $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$ be a Turing machine. Then $L(M)$ is the set of strings w in Σ^* such that $q_0 w \stackrel{*}{\vdash} \alpha p \beta$ for some state p in F and any tape strings α and β. This definition was assumed when we discussed the Turing machine of Example 8.2, which accepts strings of the form $0^n 1^n$.

The set of languages we can accept using a Turing machine is often called the *recursively enumerable languages* or RE languages. The term "recursively enumerable" comes from computational formalisms that predate the Turing machine but that define the same class of languages or arithmetic functions. We discuss the origins of the term as an aside (box) in Section 9.2.1.

8.2.6 Turing Machines and Halting

There is another notion of "acceptance" that is commonly used for Turing machines: acceptance by halting. We say a TM *halts* if it enters a state q, scanning a tape symbol X, and there is no move in this situation; i.e., $\delta(q, X)$ is undefined.

Example 8.5: The Turing machine M of Example 8.4 was not designed to accept a language; rather we viewed it as computing an arithmetic function. Note, however, that M halts on all strings of 0's and 1's, since no matter what string M finds on its tape, it will eventually cancel its second group of 0's, if it can find such a group, against its first group of 0's, and thus must reach state q_6 and halt. □

Notational Conventions for Turing Machines

The symbols we normally use for Turing machines resemble those for the other kinds of automata we have seen.

1. Lower-case letters at the beginning of the alphabet stand for input symbols.
2. Capital letters, typically near the end of the alphabet, are used for tape symbols that may or may not be input symbols. However, B is generally used for the blank symbol.
3. Lower-case letters near the end of the alphabet are strings of input symbols.
4. Greek letters are strings of tape symbols.
5. Letters such as q, p, and nearby letters are states.

We can always assume that a TM halts if it accepts. That is, without changing the language accepted, we can make $\delta(q, X)$ undefined whenever q is an accepting state. In general, without otherwise stating so:

- We assume that a TM always halts when it is in an accepting state.

Unfortunately, it is not always possible to require that a TM halts even if it does not accept. Those languages with Turing machines that do halt eventually, regardless of whether or not they accept, are called *recursive*, and we shall consider their important properties starting in Section 9.2.1. Turing machines that always halt, regardless of whether or not they accept, are a good model of an "algorithm." If an algorithm to solve a given problem exists, then we say the problem is "decidable," so TM's that always halt figure importantly into decidability theory in Chapter 9.

8.2.7 Exercises for Section 8.2

Exercise 8.2.1: Show the ID's of the Turing machine of Fig. 8.9 if the input tape contains:

***** a) 00.

b) 000111.

c) 00111.

! Exercise 8.2.2: Design Turing machines for the following languages:

* a) The set of strings with an equal number of 0's and 1's.

b) $\{a^n b^n c^n \mid n \geq 1\}$.

c) $\{ww^R \mid w$ is any string of 0's and 1's$\}$.

Exercise 8.2.3: Design a Turing machine that takes as input a number N and adds 1 to it in binary. To be precise, the tape initially contains a \$ followed by N in binary. The tape head is initially scanning the \$ in state q_0. Your TM should halt with N+1, in binary, on its tape, scanning the leftmost symbol of $N+1$, in state q_f. You may destroy the \$ in creating $N+1$, if necessary. For instance, $q_0\$10011 \vdash^* \$q_f10100$, and $q_0\$11111 \vdash^* q_f100000$.

a) Give the transitions of your Turing machine, and explain the purpose of each state.

b) Show the sequence of ID's of your TM when given input \$111.

*! **Exercise 8.2.4:** In this exercise we explore the equivalence between function computation and language recognition for Turing machines. For simplicity, we shall consider only functions from nonnegative integers to nonnegative integers, but the ideas of this problem apply to any computable functions. Here are the two central definitions:

- Define the *graph* of a function f to be the set of all strings of the form $[x, f(x)]$, where x is a nonnegative integer in binary, and $f(x)$ is the value of function f with argument x, also written in binary.

- A Turing machine is said to *compute* function f if, started with any nonnegative integer x on its tape, in binary, it halts (in any state) with $f(x)$, in binary, on its tape.

Answer the following, with informal, but clear constructions.

a) Show how, given a TM that computes f, you can construct a TM that accepts the graph of f as a language.

b) Show how, given a TM that accepts the graph of f, you can construct a TM that computes f.

c) A function is said to be *partial* if it may be undefined for some arguments. If we extend the ideas of this exercise to partial functions, then we do not require that the TM computing f halts if its input x is one of the integers for which $f(x)$ is not defined. Do your constructions for parts (a) and (b) work if the function f is partial? If not, explain how you could modify the construction to make it work.

Exercise 8.2.5: Consider the Turing machine

$$M = (\{q_0, q_1, q_2, q_f\}, \{0,1\}, \{0,1,B\}, \delta, q_0, B, \{q_f\})$$

Informally but clearly describe the language $L(M)$ if δ consists of the following sets of rules:

* a) $\delta(q_0, 0) = (q_1, 1, R)$; $\delta(q_1, 1) = (q_0, 0, R)$; $\delta(q_1, B) = (q_f, B, R)$.

b) $\delta(q_0, 0) = (q_0, B, R)$; $\delta(q_0, 1) = (q_1, B, R)$; $\delta(q_1, 1) = (q_1, B, R)$; $\delta(q_1, B) = (q_f, B, R)$.

! c) $\delta(q_0, 0) = (q_1, 1, R)$; $\delta(q_1, 1) = (q_2, 0, L)$; $\delta(q_2, 1) = (q_0, 1, R)$; $\delta(q_1, B) = (q_f, B, R)$.

8.3 Programming Techniques for Turing Machines

Our goal is to give you a sense of how a Turing machine can be used to compute in a manner not unlike that of a conventional computer. Eventually, we want to convince you that a TM is exactly as powerful as a conventional computer. In particular, we shall learn that the Turing machine can perform the sort of calculations on other Turing machines that we saw performed in Section 8.1.2 by a program that examined other programs. This "introspective" ability of both Turing machines and computer programs is what enables us to prove problems undecidable.

To make the ability of a TM clearer, we shall present a number of examples of how we might think of the tape and finite control of the Turing machine. None of these tricks extend the basic model of the TM; they are only notational conveniences. Later, we shall use them to simulate extended Turing-machine models that have additional features — for instance, more than one tape — by the basic TM model.

8.3.1 Storage in the State

We can use the finite control not only to represent a position in the "program" of the Turing machine, but to hold a finite amount of data. Figure 8.13 suggests this technique (as well as another idea: multiple tracks). There, we see the finite control consisting of not only a "control" state q, but three data elements A, B, and C. The technique requires no extension to the TM model; we merely think of the state as a tuple. In the case of Fig. 8.13, we should think of the state as $[q, A, B, C]$. Regarding states this way allows us to describe transitions in a more systematic way, often making the strategy behind the TM program more transparent.

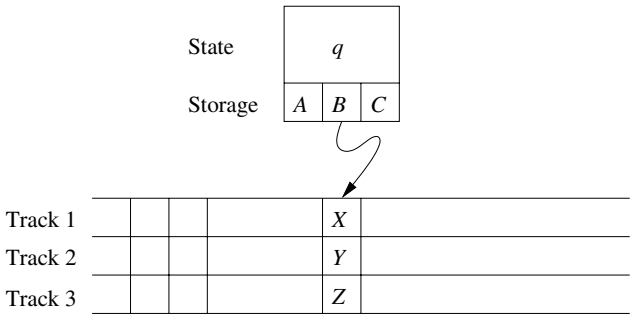

Figure 8.13: A Turing machine viewed as having finite-control storage and multiple tracks

Example 8.6 : We shall design a TM

$$M = (Q, \{0,1\}, \{0,1,B\}, \delta, [q_0, B], B, \{[q_1, B]\})$$

that remembers in its finite control the first symbol (0 or 1) that it sees, and checks that it does not appear elsewhere on its input. Thus, M accepts the language $\mathbf{01}^*+\mathbf{10}^*$. Accepting regular languages such as this one does not stress the ability of Turing machines, but it will serve as a simple demonstration.

The set of states Q is $\{q_0, q_1\} \times \{0, 1, B\}$. That is, the states may be thought of as pairs with two components:

a) A control portion, q_0 or q_1, that remembers what the TM is doing. Control state q_0 indicates that M has not yet read its first symbol, while q_1 indicates that it *has* read the symbol, and is checking that it does not appear elsewhere, by moving right and hoping to reach a blank cell.

b) A data portion, which remembers the first symbol seen, which must be 0 or 1. The symbol B in this component means that no symbol has been read.

The transition function δ of M is as follows:

1. $\delta([q_0, B], a) = ([q_1, a], a, R)$ for $a = 0$ or $a = 1$. Initially, q_0 is the control state, and the data portion of the state is B. The symbol scanned is copied into the second component of the state, and M moves right, entering control state q_1 as it does so.

2. $\delta([q_1, a], \overline{a}) = ([q_1, a], \overline{a}, R)$ where $\overline{a}$ is the "complement" of a, that is, 0 if $a = 1$ and 1 if $a = 0$. In state q_1, M skips over each symbol 0 or 1 that is different from the one it has stored in its state, and continues moving right.

3. $\delta([q_1, a], B) = ([q_1, B], B, R)$ for $a = 0$ or $a = 1$. If M reaches the first blank, it enters the accepting state $[q_1, B]$.

Notice that M has no definition for $\delta([q_1, a], a)$ for $a = 0$ or $a = 1$. Thus, if M encounters a second occurrence of the symbol it stored initially in its finite control, it halts without having entered the accepting state. □

8.3.2 Multiple Tracks

Another useful "trick" is to think of the tape of a Turing machine as composed of several tracks. Each track can hold one symbol, and the tape alphabet of the TM consists of tuples, with one component for each "track." Thus, for instance, the cell scanned by the tape head in Fig. 8.13 contains the symbol $[X, Y, Z]$. Like the technique of storage in the finite control, using multiple tracks does not extend what the Turing machine can do. It is simply a way to view tape symbols and to imagine that they have a useful structure.

Example 8.7: A common use of multiple tracks is to treat one track as holding the data and a second track as holding a mark. We can check off each symbol as we "use" it, or we can keep track of a small number of positions within the data by marking only those positions. Examples 8.2 and 8.4 were two instances of this technique, but in neither example did we think explicitly of the tape as if it were composed of tracks. In the present example, we shall use a second track explicitly to recognize the non-context-free language

$$L_{wcw} = \{wcw \mid w \text{ is in } (\mathbf{0} + \mathbf{1})^+\}$$

The Turing machine we shall design is:

$$M = (Q, \Sigma, \Gamma, \delta, [q_1, B], [B, B], \{[q_9, B]\})$$

where:

- Q: The set of states is $\{q_1, q_2, \ldots, q_9\} \times \{0, 1, B\}$, that is, pairs consisting of a control state q_i and a data component: 0, 1, or blank. We again use the technique of storage in the finite control, as we allow the state to remember an input symbol 0 or 1.

- Γ: The set of tape symbols is $\{B, *\} \times \{0, 1, c, B\}$. The first component, or track, can be either blank or "checked," represented by the symbols B and $*$, respectively. We use the $*$ to check off symbols of the first and second groups of 0's and 1's, eventually confirming that the string to the left of the center marker c is the same as the string to its right. The second component of the tape symbol is what we think of as the tape symbol itself. That is, we may think of the symbol $[B, X]$ as if it were the tape symbol X, for $X = 0, 1, c, B$.

- Σ: The input symbols are $[B, 0]$, $[B, 1]$, and $[B, c]$, which, as just mentioned, we identify with 0, 1, and c, respectively.

δ: The transition function δ is defined by the following rules, in which a and b each may stand for either 0 or 1.

1. $\delta([q_1, B], [B, a]) = ([q_2, a], [*, a], R)$. In the initial state, M picks up the symbol a (which can be either 0 or 1), stores it in its finite control, goes to control state q_2, "checks off" the symbol it just scanned, and moves right. Notice that by changing the first component of the tape symbol from B to $*$, it performs the check-off.
2. $\delta([q_2, a], [B, b]) = ([q_2, a], [B, b], R)$. M moves right, looking for the symbol c. Remember that a and b can each be either 0 or 1, independently, but cannot be c.
3. $\delta([q_2, a], [B, c]) = ([q_3, a], [B, c], R)$. When M finds the c, it continues to move right, but changes to control state q_3.
4. $\delta([q_3, a], [*, b]) = ([q_3, a], [*, b], R)$. In state q_3, M continues past all checked symbols.
5. $\delta([q_3, a], [B, a]) = ([q_4, B], [*, a], L)$. If the first unchecked symbol that M finds is the same as the symbol in its finite control, it checks this symbol, because it has matched the corresponding symbol from the first block of 0's and 1's. M goes to control state q_4, dropping the symbol from its finite control, and starts moving left.
6. $\delta([q_4, B], [*, a]) = ([q_4, B], [*, a], L)$. M moves left over checked symbols.
7. $\delta([q_4, B], [B, c]) = ([q_5, B], [B, c], L)$. When M encounters the symbol c, it switches to state q_5 and continues left. In state q_5, M must make a decision, depending on whether or not the symbol immediately to the left of the c is checked or unchecked. If checked, then we have already considered the entire first block of 0's and 1's — those to the left of the c. We must make sure that all the 0's and 1's to the right of the c are also checked, and accept if no unchecked symbols remain to the right of the c. If the symbol immediately to the left of the c is unchecked, we find the leftmost unchecked symbol, pick it up, and start the cycle that began in state q_1.
8. $\delta([q_5, B], [B, a]) = ([q_6, B], [B, a], L)$. This branch covers the case where the symbol to the left of c is unchecked. M goes to state q_6 and continues left, looking for a checked symbol.
9. $\delta([q_6, B], [B, a]) = ([q_6, B], [B, a], L)$. As long as symbols are unchecked, M remains in state q_6 and proceeds left.
10. $\delta([q_6, B], [*, a]) = ([q_1, B], [*, a], R)$. When the checked symbol is found, M enters state q_1 and moves right to pick up the first unchecked symbol.
11. $\delta([q_5, B], [*, a]) = ([q_7, B], [*, a], R)$. Now, let us pick up the branch from state q_5 where we have just moved left from the c and find a checked symbol. We start moving right again, entering state q_7.

12. $\delta([q_7, B], [B, c]) = ([q_8, B], [B, c], R)$. In state q_7 we shall surely see the c. We enter state q_8 as we do so, and proceed right.
13. $\delta([q_8, B], [*, a]) = ([q_8, B], [*, a], R)$. M moves right in state q_8, skipping over any checked 0's or 1's that it finds.
14. $\delta([q_8, B], [B, B]) = ([q_9, B], [B, B], R)$. If M reaches a blank cell in state q_8 without encountering any unchecked 0 or 1, then M accepts. If M first finds an unchecked 0 or 1, then the blocks before and after the c do not match, and M halts without accepting.

□

8.3.3 Subroutines

As with programs in general, it helps to think of Turing machines as built from a collection of interacting components, or "subroutines." A Turing-machine subroutine is a set of states that perform some useful process. This set of states includes a start state and another state that temporarily has no moves, and that serves as the "return" state to pass control to whatever other set of states called the subroutine. The "call" of a subroutine occurs whenever there is a transition to its initial state. Since the TM has no mechanism for remembering a "return address," that is, a state to go to after it finishes, should our design of a TM call for one subroutine to be called from several states, we can make copies of the subroutine, using a new set of states for each copy. The "calls" are made to the start states of different copies of the subroutine, and each copy "returns" to a different state.

Example 8.8: We shall design a TM to implement the function "multiplication." That is, our TM will start with 0^m10^n1 on its tape, and will end with 0^{mn} on the tape. An outline of the strategy is:

1. The tape will, in general, have one nonblank string of the form $0^i10^n10^{kn}$ for some k.
2. In one basic step, we change a 0 in the first group to B and add n 0's to the last group, giving us a string of the form $0^{i-1}10^n10^{(k+1)n}$.
3. As a result, we copy the group of n 0's to the end m times, once each time we change a 0 in the first group to B. When the first group of 0's is completely changed to blanks, there will be mn 0's in the last group.
4. The final step is to change the leading 10^n1 to blanks, and we are done.

The heart of this algorithm is a subroutine, which we call `Copy`. This subroutine helps implement step (2) above, copying the block of n 0's to the end. More precisely, `Copy` converts an ID of the form $0^{m-k}1q_10^n10^{(k-1)n}$ to ID $0^{m-k}1q_50^n10^{kn}$. Figure 8.14 shows the transitions of subroutine `Copy`. This

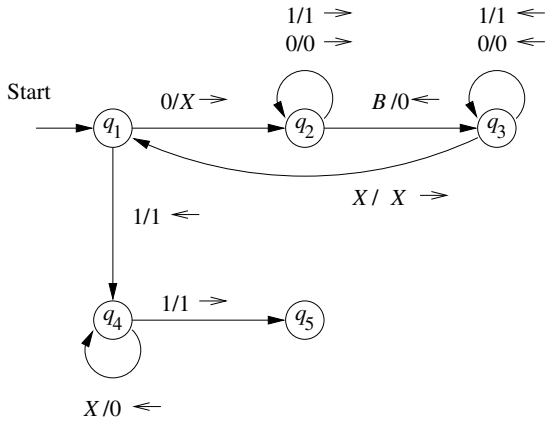

Figure 8.14: The subroutine `Copy`

subroutine marks the first 0 with an X, moves right in state q_2 until it finds a blank, copies the 0 there, and moves left in state q_3 to find the marker X. It repeats this cycle until in state q_1 it finds a 1 instead of a 0. At that point, it uses state q_4 to change the X's back to 0's, and ends in state q_5.

The complete multiplication Turing machine starts in state q_0. The first thing it does is go, in several steps, from ID $q_0 0^m 10^n$ to ID $0^{m-1}1q_1 0^n$. The transitions needed are shown in the portion of Fig. 8.15 to the left of the subroutine call; these transitions involve states q_0 and q_6 only.

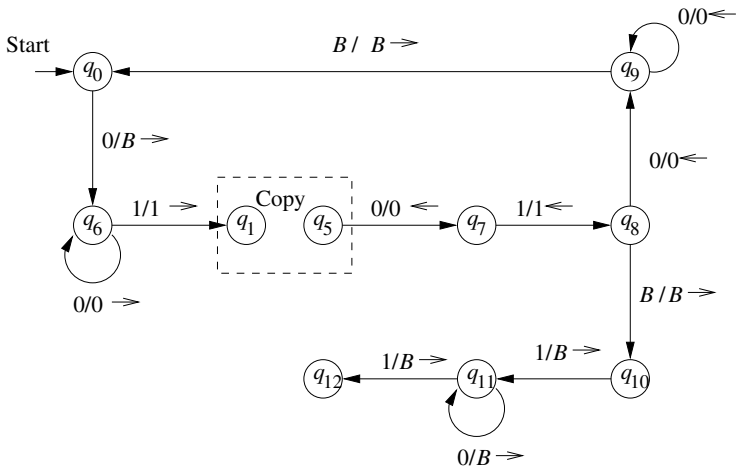

Figure 8.15: The complete multiplication program uses the subroutine `Copy`

Then, to the right of the subroutine call in Fig. 8.15 we see states q_7 through q_{12}. The purpose of states q_7, q_8, and q_9 is to take control after `Copy` has just

copied a block of n 0's, and is in ID $0^{m-k}1q_50^n10^{kn}$. Eventually, these states bring us to state $q_00^{m-k}10^n10^{kn}$. At that point, the cycle starts again, and `Copy` is called to copy the block of n 0's again.

As an exception, in state q_8 the TM may find that all m 0's have been changed to blanks (i.e., $k = m$). In that case, a transition to state q_{10} occurs. This state, with the help of state q_{11}, changes the leading 10^n1 to blanks and enters the halting state q_{12}. At this point, the TM is in ID $q_{12}0^{mn}$, and its job is done. □

8.3.4 Exercises for Section 8.3

! **Exercise 8.3.1:** Redesign your Turing machines from Exercise 8.2.2 to take advantage of the programming techniques discussed in Section 8.3.

! **Exercise 8.3.2:** A common operation in Turing-machine programs involves "shifting over." Ideally, we would like to create an extra cell at the current head position, in which we could store some character. However, we cannot edit the tape in this way. Rather, we need to move the contents of each of the cells to the right of the current head position one cell right, and then find our way back to the current head position. Show how to perform this operation. *Hint*: Leave a special symbol to mark the position to which the head must return.

* **Exercise 8.3.3:** Design a subroutine to move a TM head from its current position to the right, skipping over all 0's, until reaching a 1 or a blank. If the current position does not hold 0, then the TM should halt. You may assume that there are no tape symbols other than 0, 1, and B (blank). Then, use this subroutine to design a TM that accepts all strings of 0's and 1's that do not have two 1's in a row.

8.4 Extensions to the Basic Turing Machine

In this section we shall see certain computer models that are related to Turing machines and have the same language-recognizing power as the basic model of a TM with which we have been working. One of these, the multitape Turing machine, is important because it is much easier to see how a multitape TM can simulate real computers (or other kinds of Turing machines), compared with the single-tape model we have been studying. Yet the extra tapes add no power to the model, as far as the ability to accept languages is concerned.

We then consider the nondeterministic Turing machine, an extension of the basic model that is allowed to make any of a finite set of choices of move in a given situation. This extension also makes "programming" Turing machines easier in some circumstances, but adds no language-defining power to the basic model.

8.4.1 Multitape Turing Machines

A multitape TM is as suggested by Fig. 8.16. The device has a finite control (state), and some finite number of tapes. Each tape is divided into cells, and each cell can hold any symbol of the finite tape alphabet. As in the single-tape TM, the set of tape symbols includes a blank, and has a subset called the input symbols, of which the blank is not a member. The set of states includes an initial state and some accepting states. Initially:

1. The input, a finite sequence of input symbols, is placed on the first tape.
2. All other cells of all the tapes hold the blank.
3. The finite control is in the initial state.
4. The head of the first tape is at the left end of the input.
5. All other tape heads are at some arbitrary cell. Since tapes other than the first tape are completely blank, it does not matter where the head is placed initially; all cells of these tapes "look" the same.

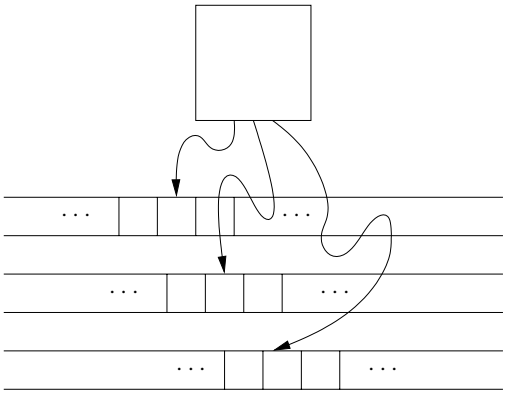

Figure 8.16: A multitape Turing machine

A move of the multitape TM depends on the state and the symbol scanned by each of the tape heads. In one move, the multitape TM does the following:

1. The control enters a new state, which could be the same as the previous state.
2. On each tape, a new tape symbol is written on the cell scanned. Any of these symbols may be the same as the symbol previously there.
3. Each of the tape heads makes a move, which can be either left, right, or stationary. The heads move independently, so different heads may move in different directions, and some may not move at all.

We shall not give the formal notation of transition rules, whose form is a straightforward generalization of the notation for the one-tape TM, except that directions are now indicated by a choice of L, R, or S. For the one-tape machine, we did not allow the head to remain stationary, so the S option was not present. You should be able to imagine an appropriate notation for instantaneous descriptions of the configuration of a multitape TM; we shall not give this notation formally. Multitape Turing machines, like one-tape TM's, accept by entering an accepting state.

8.4.2 Equivalence of One-Tape and Multitape TM's

Recall that the recursively enumerable languages are defined to be those accepted by a one-tape TM. Surely, multitape TM's accept all the recursively enumerable languages, since a one-tape TM *is* a multitape TM. However, are there languages that are not recursively enumerable, yet are accepted by multitape TM's? The answer is "no," and we prove this fact by showing how to simulate a multitape TM by a one-tape TM.

Theorem 8.9: Every language accepted by a multitape TM is recursively enumerable.

PROOF: The proof is suggested by Fig. 8.17. Suppose language L is accepted by a k-tape TM M. We simulate M with a one-tape TM N whose tape we think of as having $2k$ tracks. Half these tracks hold the tapes of M, and the other half of the tracks each hold only a single marker that indicates where the head for the corresponding tape of M is currently located. Figure 8.17 assumes $k = 2$. The second and fourth tracks hold the contents of the first and second tapes of M, track 1 holds the position of the head of tape 1, and track 3 holds the position of the second tape head.

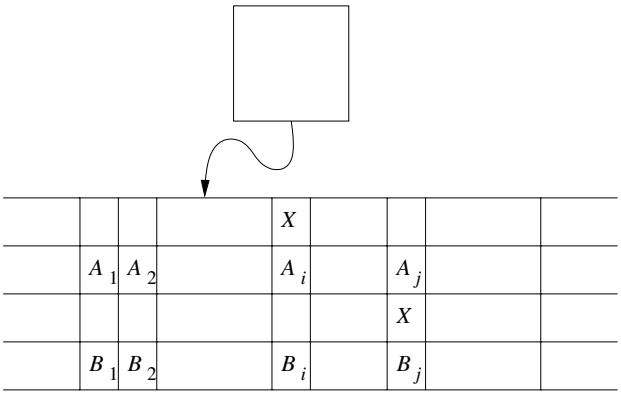

Figure 8.17: Simulation of a two-tape Turing machine by a one-tape Turing machine

A Reminder About Finiteness

A common fallacy is to confuse a value that is finite at any time with a set of values that is finite. The many-tapes-to-one construction may help us appreciate the difference. In that construction, we used tracks on the tape to record the positions of the tape heads. Why could we not store these positions as integers in the finite control? Carelessly, one could argue that after n moves, the TM can have tape head positions that must be within n positions of original head positions, and so the head only has to store integers up to n.

The problem is that, while the positions are finite at any time, the complete set of positions possible at any time is infinite. If the state is to represent any head position, then there must be a data component of the state that has any integer as value. This component forces the set of states to be infinite, even if only a finite number of them can be used at any finite time. The definition of a Turing machine requires that the *set* of states be finite. Thus, it is not permissible to store a tape-head position in the finite control.

To simulate a move of M, N's head must visit the k head markers. So that N not get lost, it must remember how many head markers are to its left at all times; that count is stored as a component of N's finite control. After visiting each head marker and storing the scanned symbol in a component of its finite control, N knows what tape symbols are being scanned by each of M's heads. N also knows the state of M, which it stores in N's own finite control. Thus, N knows what move M will make.

N now revisits each of the head markers on its tape, changes the symbol in the track representing the corresponding tapes of M, and moves the head markers left or right, if necessary. Finally, N changes the state of M as recorded in its own finite control. At this point, N has simulated one move of M.

We select as N's accepting states all those states that record M's state as one of the accepting states of M. Thus, whenever the simulated M accepts, N also accepts, and N does not accept otherwise. □

8.4.3 Running Time and the Many-Tapes-to-One Construction

Let us now introduce a concept that will become quite important later: the "time complexity" or "running time" of a Turing machine. We say the *running time* of TM M on input w is the number of steps that M makes before halting. If M doesn't halt on w, then the running time of M on w is infinite. The *time complexity* of TM M is the function $T(n)$ that is the maximum, over all inputs

w of length n, of the running time of M on w. For Turing machines that do not halt on all inputs, $T(n)$ may be infinite for some or even all n. However, we shall pay special attention to TM's that do halt on all inputs, and in particular, those that have a polynomial time complexity $T(n)$; Section 10.1 initiates this study.

The construction of Theorem 8.9 seems clumsy. In fact, the constructed one-tape TM may take much more running time than the multitape TM. However, the amounts of time taken by the two Turing machines are commensurate in a weak sense: the one-tape TM takes time that is no more than the square of the time taken by the other. While "squaring" is not a very strong guarantee, it does preserve polynomial running time. We shall see in Chapter 10 that:

a) The difference between polynomial time and higher growth rates in running time is really the divide between what we can solve by computer and what is in practice not solvable.

b) Despite extensive research, the running time needed to solve many problems has not been resolved closer than to within some polynomial. Thus, the question of whether we are using a one-tape or multitape TM to solve the problem is not crucial when we examine the running time needed to solve a particular problem.

The argument that the running times of the one-tape and multitape TM's are within a square of each other is as follows.

Theorem 8.10: The time taken by the one-tape TM N of Theorem 8.9 to simulate n moves of the k-tape TM M is $O(n^2)$.

PROOF: After n moves of M, the tape head markers cannot have separated by more than $2n$ cells. Thus, if N starts at the leftmost marker, it has to move no more than $2n$ cells right, to find all the head markers. It can then make an excursion leftward, changing the contents of the simulated tapes of M, and moving head markers left or right as needed. Doing so requires no more than $2n$ moves left, plus at most $2k$ moves to reverse direction and write a marker X in the cell to the right (in the case that a tape head of M moves right).

Thus, the number of moves by N needed to simulate one of the first n moves is no more than $4n + 2k$. Since k is a constant, independent of the number of moves simulated, this number of moves is $O(n)$. To simulate n moves requires no more than n times this amount, or $O(n^2)$. □

8.4.4 Nondeterministic Turing Machines

A *nondeterministic* Turing machine (*NTM*) differs from the deterministic variety we have been studying by having a transition function δ such that for each state q and tape symbol X, $\delta(q, X)$ is a set of triples

$$\{(q_1, Y_1, D_1),\ (q_2, Y_2, D_2), \ldots, (q_k, Y_k, D_k)\}$$

where k is any finite integer. The NTM can choose, at each step, any of the triples to be the next move. It cannot, however, pick a state from one, a tape symbol from another, and the direction from yet another.

The language accepted by an NTM M is defined in the expected manner, in analogy with the other nondeterministic devices, such as NFA's and PDA's, that we have studied. That is, M accepts an input w if there is any sequence of choices of move that leads from the initial ID with w as input, to an ID with an accepting state. The existence of other choices that do *not* lead to an accepting state is irrelevant, as it is for the NFA or PDA.

The NTM's accept no languages not accepted by a deterministic TM (or *DTM* if we need to emphasize that it is deterministic). The proof involves showing that for every NTM M_N, we can construct a DTM M_D that explores the ID's that M_N can reach by any sequence of its choices. If M_D finds one that has an accepting state, then M_D enters an accepting state of its own. M_D must be systematic, putting new ID's on a queue, rather than a stack, so that after some finite time M_D has simulated all sequences of up to k moves of M_N, for $k = 1, 2, \ldots$.

Theorem 8.11: If M_N is a nondeterministic Turing machine, then there is a deterministic Turing machine M_D such that $L(M_N) = L(M_D)$.

PROOF: M_D will be designed as a multitape TM, sketched in Fig. 8.18. The first tape of M_D holds a sequence of ID's of M_N, including the state of M_N. One ID of M_N is marked as the "current" ID, whose successor ID's are in the process of being discovered. In Fig. 8.18, the third ID is marked by an x along with the inter-ID separator, which is the $*$. All ID's to the left of the current one have been explored and can be ignored subsequently.

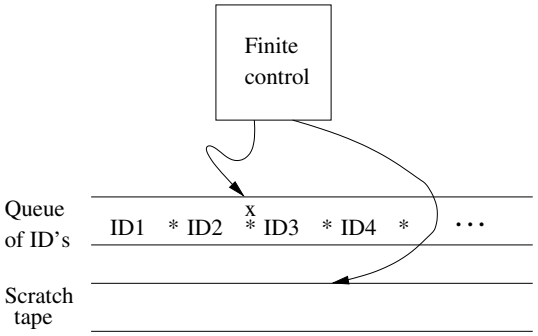

Figure 8.18: Simulation of an NTM by a DTM

To process the current ID, M_D does the following:

1. M_D examines the state and scanned symbol of the current ID. Built into the finite control of M_D is the knowledge of what choices of move M_N

has for each state and symbol. If the state in the current ID is accepting, then M_D accepts and simulates M_N no further.

2. However, if the state is not accepting, and the state-symbol combination has k moves, then M_D uses its second tape to copy the ID and then make k copies of that ID at the end of the sequence of ID's on tape 1.

3. M_D modifies each of those k ID's according to a different one of the k choices of move that M_N has from its current ID.

4. M_D returns to the marked, current ID, erases the mark, and moves the mark to the next ID to the right. The cycle then repeats with step (1).

It should be clear that the simulation is accurate, in the sense that M_D will only accept if it finds that M_N can enter an accepting ID. However, we need to confirm that if M_N enters an accepting ID after a sequence of n of its own moves, then M_D will eventually make that ID the current ID and will accept.

Suppose that m is the maximum number of choices M_N has in any configuration. Then there is one initial ID of M_N, at most m ID's that M_N can reach after one move, at most m^2 ID's M_N can reach after two moves, and so on. Thus, after n moves, M_N can reach at most $1 + m + m^2 + \cdots + m^n$ ID's. This number is at most nm^n ID's.

The order in which M_D explores ID's of M_N is "breadth first"; that is, it explores all ID's reachable by 0 moves (i.e., the initial ID), then all ID's reachable by one move, then those reachable by two moves, and so on. In particular, M_D will make current, and consider the successors of, all ID's reachable by up to n moves before considering any ID's that are only reachable by more than n moves.

As a consequence, the accepting ID of M_N will be considered by M_D among the first nm^n ID's that it considers. We only care that M_D considers this ID in some finite time, and this bound is sufficient to assure us that the accepting ID is considered eventually. Thus, if M_N accepts, then so does M_D. Since we already observed that if M_D accepts it does so only because M_N accepts, we conclude that $L(M_N) = L(M_D)$. □

Notice that the constructed deterministic TM may take exponentially more time than the nondeterministic TM. It is unknown whether or not this exponential slowdown is necessary. In fact, Chapter 10 is devoted to this question and the consequences of someone discovering a better way to simulate NTM's deterministically.

8.4.5 Exercises for Section 8.4

Exercise 8.4.1: Informally but clearly describe multitape Turing machines that accept each of the languages of Exercise 8.2.2. Try to make each of your Turing machines run in time proportional to the input length.

Exercise 8.4.2: Here is the transition function of a nondeterministic TM $M = (\{q_0, q_1, q_2\}, \{0,1\}, \{0,1,B\}, \delta, q_0, B, \{q_2\})$:

δ	0	1	B
q_0	$\{(q_0, 1, R)\}$	$\{(q_1, 0, R)\}$	$\emptyset$
q_1	$\{(q_1, 0, R),\ (q_0, 0, L)\}$	$\{(q_1, 1, R),\ (q_0, 1, L)\}$	$\{(q_2, B, R)\}$
q_2	$\emptyset$	$\emptyset$	$\emptyset$

Show the ID's reachable from the initial ID if the input is:

* a) 01.

b) 011.

! **Exercise 8.4.3:** Informally but clearly describe nondeterministic Turing machines — multitape if you like — that accept the following languages. Try to take advantage of nondeterminism to avoid iteration and save time in the nondeterministic sense. That is, prefer to have your NTM branch a lot, while each branch is short.

* a) The language of all strings of 0's and 1's that have some string of length 100 that repeats, not necessarily consecutively. Formally, this language is the set of strings of 0's and 1's of the form $wxyxz$, where $|x| = 100$, and w, y, and z are of arbitrary length.

b) The language of all strings of the form $w_1 \# w_2 \# \cdots \# w_n$, for any n, such that each w_i is a string of 0's and 1's, and for some j, w_j is the integer j in binary.

c) The language of all strings of the same form as (b), but for at least two values of j, we have w_j equal to j in binary.

! **Exercise 8.4.4:** Consider the nondeterministic Turing machine

$$M = (\{q_0, q_1, q_2, q_f\}, \{0,1\}, \{0,1,B\}, \delta, q_0, B, \{q_f\})$$

Informally but clearly describe the language $L(M)$ if δ consists of the following sets of rules: $\delta(q_0, 0) = \{(q_0, 1, R),\ (q_1, 1, R)\}$; $\delta(q_1, 1) = \{(q_2, 0, L)\}$; $\delta(q_2, 1) = \{(q_0, 1, R)\}$; $\delta(q_1, B) = \{(q_f, B, R)\}$.

* **Exercise 8.4.5:** Consider a nondeterministic TM whose tape is infinite in both directions. At some time, the tape is completely blank, except for one cell, which holds the symbol \$. The head is currently at some blank cell, and the state is q.

a) Write transitions that will enable the NTM to enter state p, scanning the \$.

! b) Suppose the TM were deterministic instead. How would you enable it to find the \$ and enter state p?

Exercise 8.4.6: Design the following 2-tape TM to accept the language of all strings of 0's and 1's with an equal number of each. The first tape contains the input, and is scanned from left to right. The second tape is used to store the excess of 0's over 1's, or vice-versa, in the part of the input seen so far. Specify the states, transitions, and the intuitive purpose of each state.

Exercise 8.4.7: In this exercise, we shall implement a stack using a special 3-tape TM.

1. The first tape will be used only to hold and read the input. The input alphabet consists of the symbol $\uparrow$, which we shall interpret as "pop the stack," and the symbols a and b, which are interpreted as "push an a (respectively b) onto the stack."

2. The second tape is used to store the stack.

3. The third tape is the output tape. Every time a symbol is popped from the stack, it must be written on the output tape, following all previously written symbols.

The Turing machine is required to start with an empty stack and implement the sequence of push and pop operations, as specified on the input, reading from left to right. If the input causes the TM to try to pop and empty stack, then it must halt in a special error state q_e. If the entire input leaves the stack empty at the end, then the input is accepted by going to the final state q_f. Describe the transition function of the TM informally but clearly. Also, give a summary of the purpose of each state you use.

Exercise 8.4.8: In Fig. 8.17 we saw an example of the general simulation of a k-tape TM by a one-tape TM.

***** a) Suppose this technique is used to simulate a 5-tape TM that had a tape alphabet of seven symbols. How many tape symbols would the one-tape TM have?

***** b) An alternative way to simulate k tapes by one is to use a $(k+1)$st track to hold the head positions of all k tapes, while the first k tracks simulate the k tapes in the obvious manner. Note that in the $(k+1)$st track, we must be careful to distinguish among the tape heads and to allow for the possibility that two or more heads are at the same cell. Does this method reduce the number of tape symbols needed for the one-tape TM?

c) Another way to simulate k tapes by 1 is to avoid storing the head positions altogether. Rather, a $(k+1)$st track is used only to mark one cell of the tape. At all times, each simulated tape is positioned on its track so the head is at the marked cell. If the k-tape TM moves the head of tape i, then the simulating one-tape TM slides the entire nonblank contents of the ith track one cell in the opposite direction, so the marked cell continues to

hold the cell scanned by the ith tape head of the k-tape TM. Does this method help reduce the number of tape symbols of the one-tape TM? Does it have any drawbacks compared with the other methods discussed?

! **Exercise 8.4.9:** A *k-head* Turing machine has k heads reading cells of one tape. A move of this TM depends on the state and on the symbol scanned by each head. In one move, the TM can change state, write a new symbol on the cell scanned by each head, and can move each head left, right, or keep it stationary. Since several heads may be scanning the same cell, we assume the heads are numbered 1 through k, and the symbol written by the highest numbered head scanning a given cell is the one that actually gets written there. Prove that the languages accepted by k-head Turing machines are the same as those accepted by ordinary TM's.

!! **Exercise 8.4.10:** A *two-dimensional* Turing machine has the usual finite-state control but a tape that is a two-dimensional grid of cells, infinite in all directions. The input is placed on one row of the grid, with the head at the left end of the input and the control in the start state, as usual. Acceptance is by entering a final state, also as usual. Prove that the languages accepted by two-dimensional Turing machines are the same as those accepted by ordinary TM's.

8.5 Restricted Turing Machines

We have seen seeming generalizations of the Turing machine that do not add any language-recognizing power. Now, we shall consider some examples of apparent restrictions on the TM that also give exactly the same language-recognizing power. Our first restriction is minor but useful in a number of constructions to be seen later: we replace the TM tape that is infinite in both directions by a tape that is infinite only to the right. We also forbid this restricted TM to print a blank as the replacement tape symbol. The value of these restrictions is that we can assume ID's consist of only nonblank symbols, and that they always begin at the left end of the input.

We then explore certain kinds of multitape Turing machines that are generalized pushdown automata. First, we restrict the tapes of the TM to behave like stacks. Then, we further restrict the tapes to be "counters," that is, they can only represent one integer, and the TM can only distinguish a count of 0 from any nonzero count. The impact of this discussion is that there are several very simple kinds of automata that have the full power of any computer. Moreover, undecidable problems about Turing machines, which we see in Chapter 9, apply as well to these simple machines.

8.5.1 Turing Machines With Semi-infinite Tapes

While we have allowed the tape head of a Turing machine to move either left or right from its initial position, it is only necessary that the TM's head be

allowed to move within the positions at and to the right of the initial head position. In fact, we can assume the tape is *semi-infinite*, that is, there are no cells to the left of the initial head position. In the next theorem, we shall give a construction that shows a TM with a semi-infinite tape can simulate one whose tape is, like our original TM model, infinite in both directions.

The trick behind the construction is to use two tracks on the semi-infinite tape. The upper track represents the cells of the original TM that are at or to the right of the initial head position. The lower track represents the positions left of the initial position, but in reverse order. The exact arrangement is suggested in Fig. 8.19. The upper track represents cells $X_0, X_1, \ldots$, where X_0 is the initial position of the head; X_1, X_2, and so on, are the cells to its right. Cells X_{-1}, X_{-2}, and so on, represent cells to the left of the initial position. Notice the * on the leftmost cell's bottom track. This symbol serves as an endmarker and prevents the head of the semi-infinite TM from accidentally falling off the left end of the tape.

X_0	X_1	X_2	$\ldots$
*	X_{-1}	X_{-2}	$\ldots$

Figure 8.19: A semi-infinite tape can simulate a two-way infinite tape

We shall make one more restriction to our Turing machine: it never writes a blank. This simple restriction, coupled with the restriction that the tape is only semi-infinite, means that the tape is at all times a prefix of nonblank symbols followed by an infinity of blanks. Further, the sequence of nonblanks always begins at the initial tape position. We shall see in Theorem 9.19, and again in Theorem 10.9, how useful it is to assume ID's have this form.

Theorem 8.12: Every language accepted by a TM M_2 is also accepted by a TM M_1 with the following restrictions:

1. M_1's head never moves left of its initial position.
2. M_1 never writes a blank.

PROOF: Condition (2) is quite easy. Create a new tape symbol B' that functions as a blank, but is not the blank B. That is:

a) If M_2 has a rule $\delta_2(q, X) = (p, B, D)$, change this rule to $\delta_2(q, X) = (p, B', D)$.

b) Then, let $\delta_2(q, B')$ be the same as $\delta_2(q, B)$, for every state q.

Condition (1) requires more effort. Let

$$M_2 = (Q_2, \Sigma, \Gamma_2, \delta_2, q_2, B, F_2)$$

be the TM M_2 as modified above, so it never writes the blank B. Construct

$$M_1 = (Q_1, \Sigma \times \{B\}, \Gamma_1, \delta_1, q_0, [B, B], F_1)$$

where:

Q_1: The states of M_1 are $\{q_0, q_1\} \cup \big(Q_2 \times \{U, L\}\big)$. That is, the states of M_1 are the initial state q_0 another state q_1, and all the states of M_2 with a second data component that is either U or L (upper or lower). The second component tells us whether the upper or lower track, as in Fig. 8.19 is being scanned by M_2. Put another way, U means the head of M_2 is at or to the right of its initial position, and L means it is to the left of that position.

Γ_1: The tape symbols of M_1 are all pairs of symbols from Γ_2, that is, $\Gamma_2 \times \Gamma_2$. The input symbols of M_1 are those pairs with an input symbol of M_2 in the first component and a blank in the second component, that is, pairs of the form $[a, B]$, where a is in Σ. The blank of M_1 has blanks in both components. Additionally, for every symbol X in Γ_2, there is a pair $[X, *]$ in Γ_1. Here, * is a new symbol, not in Γ_2, and serves to mark the left end of M_1's tape.

δ_1: The transitions of M_1 are as follows:

1. $\delta_1(q_0, [a, B]) = (q_1, [a, *], R)$, for any a in Σ. The first move of M_1 puts the * marker in the lower track of the leftmost cell. The state becomes q_1, and the head moves right, because it cannot move left or remain stationary.
2. $\delta_1(q_1, [X, B]) = ([q_2, U], [X, B], L)$, for any X in Γ_2. In state q_1, M_1 establishes the initial conditions of M_2, by returning the head to its initial position and changing the state to $[q_2, U]$, i.e., the initial state of M_2, with attention focused on the upper track of M_1.
3. If $\delta_2(q, X) = (p, Y, D)$, then for every Z in Γ_2:
   (a) $\delta_1([q, U], [X, Z]) = ([p, U], [Y, Z], D)$ and
   (b) $\delta_1([q, L], [Z, X]) = ([p, L], [Z, Y], \overline{D})$,
   where $\overline{D}$ is the direction opposite D, that is, L if $D = R$ and R if $D = L$. If M_1 is not at its leftmost cell, then it simulates M_2 on the appropriate track — the upper track if the second component of state is U and the lower track if the second component is L. Note, however, that when working on the lower track, M_2 moves in the direction opposite that of M_2. That choice makes sense, because the left half of M_2's tape has been folded, in reverse, along the lower track of M_1's tape.
4. If $\delta_2(q, X) = (p, Y, R)$, then

$$\delta_1([q, L], [X, *]) = \delta_1([q, U], [X, *]) = ([p, U], [Y, *], R)$$

This rule covers one case of how the left endmarker * is handled. If M_2 moves right from its initial position, then regardless of whether it had previously been to the left or the right of that position (as reflected in the fact that the second component of M_1's state could be L or U), M_1 must move right and focus on the upper track. That is, M_1 will next be at the position represented by X_1 in Fig. 8.19.

5. If $\delta_2(q, X) = (p, Y, L)$, then

$$\delta_1([q, L], [X, *]) = \delta_1([q, U], [X, *]) = ([p, L], [Y, *], R)$$

This rule is similar to the previous, but covers the case where M_2 moves left from its initial position. M_1 must move right from its endmarker, but now focuses on the lower track, i.e., the cell indicated by X_{-1} in Fig. 8.19.

F_1: The accepting states F_1 are those states in $F_2 \times \{U, L\}$, that is all states of M_1 whose first component is an accepting state of M_2. The attention of M_1 may be focused on either the upper or lower track at the time it accepts.

The proof of the theorem is now essentially complete. We may observe by induction on the number of moves made by M_2 that M_1 will mimic the ID of M_2 on its own tape, if you take the lower track, reverse it, and follow it by the upper track. Also, we note that M_1 enters one of its accepting states exactly when M_2 does. Thus, $L(M_1) = L(M_2)$. □

8.5.2 Multistack Machines

We now consider several computing models that are based on generalizations of the pushdown automaton. First, we consider what happens when we give the PDA several stacks. We already know, from Example 8.7, that a Turing machine can accept languages that are not accepted by any PDA with one stack. It turns out that if we give the PDA two stacks, then it can accept any language that a TM can accept.

We shall then consider a class of machines called "counter machines." These machines have only the ability to store a finite number of integers ("counters"), and to make different moves depending on which, if any, of the counters are currently 0. The counter machine can only add or subtract one from the counter, and cannot tell two different nonzero counts from each other. In effect, a counter is like a stack on which we can place only two symbols: a bottom-of-stack marker that appears only at the bottom, and one other symbol that may be pushed and popped from the stack.

We shall not give a formal treatment of the multistack machine, but the idea is suggested by Fig. 8.20. A k-stack machine is a deterministic PDA with k stacks. It obtains its input, like the PDA does, from an input source, rather than having the input placed on a tape or stack, as the TM does. The multistack

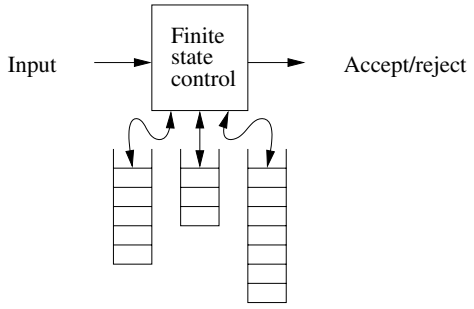

Figure 8.20: A machine with three stacks

machine has a finite control, which is in one of a finite set of states. It has a finite stack alphabet, which it uses for all its stacks. A move of the multistack machine is based on:

1. The state of the finite control.
2. The input symbol read, which is chosen from the finite input alphabet. Alternatively, the multistack machine can make a move using ϵ input, but to make the machine deterministic, there cannot be a choice of an ϵ-move or a non-ϵ-move in any situation.
3. The top stack symbol on each of its stacks.

In one move, the multistack machine can:

a) Change to a new state.

b) Replace the top symbol of each stack with a string of zero or more stack symbols. There can be (and usually is) a different replacement string for each stack.

Thus, a typical transition rule for a k-stack machine looks like:

$$\delta(q, a, X_1, X_2, \ldots, X_k) = (p, \gamma_1, \gamma_2, \ldots, \gamma_k)$$

The interpretation of this rule is that in state q, with X_i on top of the ith stack, for $i = 1, 2, \ldots, k$, the machine may consume a (either an input symbol or ϵ) from its input, go to state p, and replace X_i on top of the ith stack by string γ_i, for each $i = 1, 2, \ldots, k$. The multistack machine accepts by entering a final state.

We add one capability that simplifies input processing by this deterministic machine: we assume there is a special symbol \$, called the *endmarker*, that appears only at the end of the input and is not part of that input. The presence of the endmarker allows us to know when we have consumed all the available

input. We shall see in the next theorem how the endmarker makes it easy for the multistack machine to simulate a Turing machine. Notice that the conventional TM needs no special endmarker, because the first blank serves to mark the end of the input.

Theorem 8.13: If a language L is accepted by a Turing machine, then L is accepted by a two-stack machine.

PROOF: The essential idea is that two stacks can simulate one Turing-machine tape, with one stack holding what is to the left of the head and the other stack holding what is to the right of the head, except for the infinite strings of blanks beyond the leftmost and rightmost nonblanks. In more detail, let L be $L(M)$ for some (one-tape) TM M. Our two-stack machine S will do the following:

1. S begins with a *bottom-of-stack marker* on each stack. This marker can be the start symbol for the stacks, and must not appear elsewhere on the stacks. In what follows, we shall say that a "stack is empty" when it contains only the bottom-of-stack marker.

2. Suppose that $w\$$ is on the input of S. S copies w onto its first stack, ceasing to copy when it reads the endmarker on the input.

3. S pops each symbol in turn from its first stack and pushes it onto its second stack. Now, the first stack is empty, and the second stack holds w, with the left end of w at the top.

4. S enters the (simulated) start state of M. It has an empty first stack, representing the fact that M has nothing but blanks to the left of the cell scanned by its tape head. S has a second stack holding w, representing the fact that w appears at and to the right of the cell scanned by M's head.

5. S simulates a move of M as follows.

   (a) S knows the state of M, say q, because S simulates the state of M in its own finite control.

   (b) S knows the symbol X scanned by M's tape head; it is the top of S's second stack. As an exception, if the second stack has only the bottom-of-stack marker, then M has just moved to a blank; S interprets the symbol scanned by M as the blank.

   (c) Thus, S knows the next move of M.

   (d) The next state of M is recorded in a component of S's finite control, in place of the previous state.

   (e) If M replaces X by Y and moves right, then S pushes Y onto its first stack, representing the fact that Y is now to the left of M's head. X is popped off the second stack of S. However, there are two exceptions:

i. If the second stack has only a bottom-of-stack marker (and therefore, X is the blank), then the second stack is not changed; M has moved to yet another blank further to the right.

ii. If Y is blank, and the first stack is empty, then that stack remains empty. The reason is that there are still only blanks to the left of M's head.

(f) If M replaces X by Y and moves left, S pops the top of the first stack, say Z, then replaces X by ZY on the second stack. This change reflects the fact that what used to be one position left of the head is now at the head. As an exception, if Z is the bottom-of-stack marker, then M must push BY onto the second stack and not pop the first stack.

6. S accepts if the new state of M is accepting. Otherwise, S simulates another move of M in the same way.

□

8.5.3 Counter Machines

A *counter machine* may be thought of in one of two ways:

1. The counter machine has the same structure as the multistack machine (Fig. 8.20), but in place of each stack is a counter. Counters hold any nonnegative integer, but we can only distinguish between zero and nonzero counters. That is, the move of the counter machine depends on its state, input symbol, and which, if any, of the counters are zero. In one move, the counter machine can:

   (a) Change state.

   (b) Add or subtract 1 from any of its counters, independently. However, a counter is not allowed to become negative, so it cannot subtract 1 from a counter that is currently 0.

2. A counter machine may also be regarded as a restricted multistack machine. The restrictions are as follows:

   (a) There are only two stack symbols, which we shall refer to as Z_0 (the *bottom-of-stack marker*), and X.

   (b) Z_0 is initially on each stack.

   (c) We may replace Z_0 only by a string of the form X^iZ_0, for some $i \geq 0$.

   (d) We may replace X only by X^i for some $i \geq 0$. That is, Z_0 appears only on the bottom of each stack, and all other stack symbols, if any, are X.

We shall use definition (1) for counter machines, but the two definitions clearly define machines of equivalent power. The reason is that stack X^iZ_0 can be identified with the count i. In definition (2), we can tell count 0 from other counts, because for count 0 we see Z_0 on top of the stack, and otherwise we see X. However, we cannot distinguish two positive counts, since both have X on top of the stack.

8.5.4 The Power of Counter Machines

There are a few observations about the languages accepted by counter machines that are obvious but worth stating:

- Every language accepted by a counter machine is recursively enumerable. The reason is that a counter machine is a special case of a stack machine, and a stack machine is a special case of a multitape Turing machine, which accepts only recursively enumerable languages by Theorem 8.9.
- Every language accepted by a one-counter machine is a CFL. Note that a counter, in point-of-view (2), is a stack, so a one-counter machine is a special case of a one-stack machine, i.e., a PDA. In fact, the languages of one-counter machines are accepted by deterministic PDA's, although the proof is surprisingly complex. The difficulty in the proof stems from the fact that the multistack and counter machines have an endmarker \$ at the end of their input. A nondeterministic PDA can guess that it has seen the last input symbol and is about to see the \$; thus it is clear that a nondeterministic PDA without the endmarker can simulate a DPDA with the endmarker. However, the hard proof, which we shall not attack, is to show that a DPDA without the endmarker can simulate a DPDA *with* the endmarker.

The surprising result about counter machines is that two counters are enough to simulate a Turing machine and therefore to accept every recursively enumerable language. It is this result we address now, first showing that three counters are enough, and then simulating three counters by two counters.

Theorem 8.14: Every recursively enumerable language is accepted by a three-counter machine.

PROOF: Begin with Theorem 8.13, which says that every recursively enumerable language is accepted by a two-stack machine. We then need to show how to simulate a stack with counters. Suppose there are $r-1$ tape symbols used by the stack machine. We may identify the symbols with the digits 1 through $r-1$, and think of a stack $X_1X_2\cdots X_n$ as an integer in base r. That is, this stack (whose top is at the left end, as usual) is represented by the integer $X_nr^{n-1} + X_{n-1}r^{n-2} + \cdots + X_2r + X_1$.

We use two counters to hold the integers that represent each of the two stacks. The third counter is used to adjust the other two counters. In particular, we need the third counter when we either divide or multiply a count by r.

The operations on a stack can be broken into three kinds: pop the top symbol, change the top symbol, and push a symbol onto the stack. A move of the two-stack machine may involve several of these operations; in particular, replacing the top stack symbol X by a string of symbols must be broken down into replacing X and then pushing additional symbols onto the stack. We perform these operations on a stack that is represented by a count i, as follows. Note that it is possible to use the finite control of the multistack machine to do each of the operations that requires counting up to r or less.

1. To pop the stack, we must replace i by i/r, throwing away any remainder, which is X_1. Starting with the third counter at 0, we repeatedly reduce the count i by r, and increase the third counter by 1. When the counter that originally held i reaches 0, we stop. Then, we repeatedly increase the original counter by 1 and decrease the third counter by 1, until the third counter becomes 0 again. At this time, the counter that used to hold i holds i/r.

2. To change X to Y on the top of a stack that is represented by count i, we increment or decrement i by a small amount, surely no more than r. If $Y > X$, as digits, increment i by $Y - X$; if $Y < X$ then decrement i by $X - Y$.

3. To push X onto a stack that initially holds i, we need to replace i by $ir + X$. We first multiply by r. To do so, repeatedly decrement the count i by 1 and increase the third counter (which starts from 0, as always), by r. When the original counter becomes 0, we have ir on the third counter. Copy the third counter to the original counter and make the third counter 0 again, as we did in item (1). Finally, we increment the original counter by X.

To complete the construction, we must initialize the counters to simulate the stacks in their initial condition: holding only the start symbol of the two-stack machine. This step is accomplished by incrementing the two counters involved to some small integer, whichever integer from 1 to $r-1$ corresponds to the start symbol. □

Theorem 8.15: Every recursively enumerable language is accepted by a two-counter machine.

PROOF: With the previous theorem, we only have to show how to simulate three counters with two counters. The idea is to represent the three counters, say i, j, and k, by a single integer. The integer we choose is $m = 2^i 3^j 5^k$. One counter will hold this number, while the other is used to help multiply or divide m by one of the first three primes: 2, 3, and 5. To simulate the three-counter machine, we need to perform the following operations:

1. Increment i, j, and/or k. To increment i by 1, we multiply m by 2. We already saw in the proof of Theorem 8.14 how to multiply a count

Choice of Constants in the 3-to-2 Counter Construction

Notice how important it is in the proof of Theorem 8.15 2, 3, and 5 are distinct primes. If we had chosen, say $m = 2^i 3^j 4^k$, then $m = 12$ could represent either $i = 0$, $j = 1$, and $k = 1$, or it could represent $i = 2$, $j = 1$, and $k = 0$. Thus, we could not tell whether i or k was 0, and thus could not simulate the 3-counter machine reliably.

by any constant r, using a second counter. Likewise, we increment j by multiplying m by 3, and we increment k by multiplying m by 5.

2. Tell which, if any, of i, j, and k are 0. To tell if $i = 0$, we must determine whether m is divisible by 2. Copy m into the second counter, using the state of the counter machine to remember whether we have decremented m an even or odd number of times. If we have decremented m an odd number of times when it becomes 0, then $i = 0$. We then restore m by copying the second counter to the first. Similarly, we test if $j = 0$ by determining whether m is divisible by 3, and we test if $k = 0$ by determining whether m is divisible by 5.

3. Decrement i, j, and/or k. To do so, we divide m by 2, 3, or 5, respectively. The proof of Theorem 8.14 tells us how to perform the division by any constant, using an extra counter. Since the 3-counter machine cannot decrease a count below 0, it is an error, and the simulating 2-counter machine halts without accepting, if m is not evenly divisible by the constant by which we are dividing.

□

8.5.5 Exercises for Section 8.5

Exercise 8.5.1: Informally but clearly describe counter machines that accept the following languages. In each case, use as few counters as possible, but not more than two counters.

***** a) $\{0^n 1^m \mid n \geq m \geq 1\}$.

b) $\{0^n 1^m \mid m \geq n \geq 1\}$.

***!** c) $\{a^i b^j c^k \mid i = j \text{ or } i = k\}$.

!! d) $\{a^i b^j c^k \mid i = j \text{ or } i = k \text{ or } j = k\}$.

!! **Exercise 8.5.2:** The purpose of this exercise is to show that a one-stack machine with an endmarker on the input has no more power than a deterministic PDA. $L\$$ is the concatenation of language L with the language containing only the one string \$; that is, $L\$$ is the set of all strings $w\$$ such that w is in L. Show that if $L\$$ is a language accepted by a DPDA, where \$ is the endmarker symbol, not appearing in any string of L, then L is also accepted by some DPDA. *Hint*: This question is really one of showing that the DPDA languages are closed under the operation L/a defined in Exercise 4.2.2. You must modify the DPDA P for $L\$$ by replacing each of its stack symbols X by all possible pairs (X, S), where S is a set of states. If P has stack $X_1X_2\cdots X_n$, then the constructed DPDA for L has stack $(X_1, S_1)(X_2, S_2)\cdots(X_n, S_n)$, where each S_i is the set of states q such that P, started in ID $(q, a, X_iX_{i+1}\cdots X_n)$ will accept.

8.6 Turing Machines and Computers

Now, let us compare the Turing machine and the common sort of computer that we use daily. While these models appear rather different, they can accept exactly the same languages — the recursively enumerable languages. Since the notion of "a common computer" is not well defined mathematically, the arguments in this section are necessarily informal. We must appeal to your intuition about what computers can do, especially when the numbers involved exceed normal limits that are built into the architecture of these machines (e.g., 32-bit address spaces). The claims of this section can be divided into two parts:

1. A computer can simulate a Turing machine.
2. A Turing machine can simulate a computer, and can do so in an amount of time that is at most some polynomial in the number of steps taken by the computer.

8.6.1 Simulating a Turing Machine by Computer

Let us first examine how a computer can simulate a Turing machine. Given a particular TM M, we must write a program that acts like M. One aspect of M is its finite control. Since there are only a finite number of states and a finite number of transition rules, our program can encode states as character strings and use a table of transitions, which it looks up to determine each move. Likewise, the tape symbols can be encoded as character strings of a fixed length, since there are only a finite number of tape symbols.

A serious question arises when we consider how our program is to simulate the Turing-machine tape. This tape can grow infinitely long, but the computer's memory — main memory, disk, and other storage devices — are finite. Can we simulate an infinite tape with a fixed amount of memory?

If there is no opportunity to replace storage devices, then in fact we cannot; a computer would then be a finite automaton, and the only languages it could

accept would be regular. However, common computers have swappable storage devices, perhaps a "Zip" disk, for example. In fact, the typical hard disk is removable and can be replaced by an empty, but otherwise identical disk.

Since there is no obvious limit on how many disks we could use, let us assume that as many disks as the computer needs is available. We can thus arrange that the disks are placed in two stacks, as suggested by Fig. 8.21. One stack holds the data in cells of the Turing-machine tape that are located significantly to the left of the tape head, and the other stack holds data significantly to the right of the tape head. The further down the stacks, the further away from the tape head the data is.

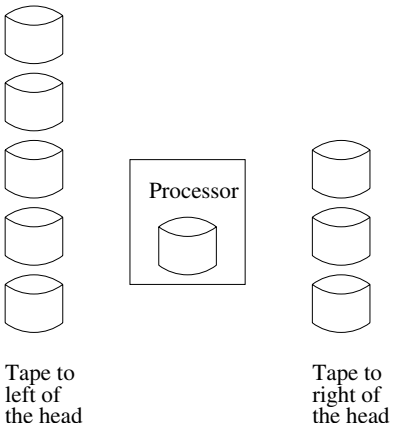

Figure 8.21: Simulating a Turing machine with a common computer

If the tape head of the TM moves sufficiently far to the left that it reaches cells that are not represented by the disk currently mounted in the computer, then it prints a message "swap left." The currently mounted disk is removed by a human operator and placed on the top of the right stack. The disk on top of the left stack is mounted in the computer, and computation resumes.

Similarly, if the TM's tape head reaches cells so far to the right that these cells are not represented by the mounted disk, then a "swap right" message is printed. The human operator moves the currently mounted disk to the top of the left stack, and mounts the disk on top of the right stack in the computer. If either stack is empty when the computer asks that a disk from that stack be mounted, then the TM has entered an all-blank region of the tape. In that case, the human operator must go to the store and buy a fresh disk to mount.

8.6.2 Simulating a Computer by a Turing Machine

We also need to consider the opposite comparison: are there things a common computer can do that a Turing machine cannot. An important subordinate question is whether the computer can do certain things much faster than a

The Problem of Very Large Tape Alphabets

The argument of Section 8.6.1 becomes questionable if the number of tape symbols is so large that the code for one tape symbol doesn't fit on a disk. There would have to be very many tape symbols indeed, since a 30 gigabyte disk, for instance, can represent any of $2^{240000000000}$ symbols. Likewise, the number of states could be so large that we could not represent the state using the entire disk.

One resolution of this problem begins by limiting the number of tape symbols a TM uses. We can always encode an arbitrary tape alphabet in binary. Thus, any TM M can be simulated by another TM M' that uses only tape symbols 0, 1, and B. However, M' needs many states, since to simulate a move of M, the TM M' must scan its tape and remember, in its finite control, all the bits that tell it what symbol M is scanning. In this manner, we are left with very large state sets, and the PC that simulates M' may have to mount and dismount several disks when deciding what the state of M' is and what the next move of M' should be. No one ever thinks about computers performing tasks of this nature, so the typical operating system has no support for a program of this type. However, if we wished, we could program the raw computer and give it this capability.

Fortunately, the question of how to simulate a TM with a huge number of states or tape symbols can be finessed. We shall see in Section 9.2.3 that one can design a TM that is in effect a "stored program" TM. This TM, called "universal," takes the transition function of any TM, encoded in binary on its tape, and simulates that TM. The universal TM has quite reasonable numbers of states and tape symbols. By simulating the universal TM, a common computer can be programmed to accept any recursively enumerable language that we wish, without having to resort to simulation of numbers of states that stress the limits of what can be stored on a disk.

Turing machine. In this section, we argue that a TM can simulate a computer, and in Section 8.6.3 we argue that the simulation can be done sufficiently fast that "only" a polynomial separates the running times of the computer and TM on a given problem. Again, let us remind the reader that there are important reasons to think of all running times that lie within a polynomial of one another to be similar, while exponential differences in running time are "too much." We take up the theory of polynomial versus exponential running times in Chapter 10.

To begin our study of how a TM simulates a computer, let us give a realistic but informal model of how a typical computer operates.

a) First, we shall suppose that the storage of a computer consists of an indef-

initely long sequence of *words*, each with an *address*. In a real computer, words might be 32 or 64 bits long, but we shall not put a limit on the length of a given word. Addresses will be assumed to be integers 0, 1, 2, and so on. In a real computer, individual bytes would be numbered by consecutive integers, so words would have addresses that are multiples of 4 or 8, but this difference is unimportant. Also, in a real computer, there would be a limit on the number of words in "memory," but since we want to account for the content of an arbitrary number of disks or other storage devices, we shall assume there is no limit to the number of words.

b) We assume that the program of the computer is stored in some of the words of memory. These words each represent a simple instruction, as in the machine or assembly language of a typical computer. Examples are instructions that move data from one word to another or that add one word to another. We assume that "indirect addressing" is permitted, so one instruction could refer to another word and use the contents of that word as the address of the word to which the operation is applied. This capability, found in all modern computers, is needed to perform array accesses, to follow links in a list, or to do pointer operations in general.

c) We assume that each instruction involves a limited (finite) number of words, and that each instruction changes the value of at most one word.

d) A typical computer has *registers*, which are memory words with especially fast access. Often, operations such as addition are restricted to occur in registers. We shall not make any such restrictions, but will allow any operation to be performed on any word. The relative speed of operations on different words will not be taken into account, nor need it be if we are only comparing the language-recognizing abilities of computers and Turing machines. Even if we are interested in running time to within a polynomial, the relative speeds of different word accesses is unimportant, since those differences are "only" a constant factor.

Figure 8.22 suggests how the Turing machine would be designed to simulate a computer. This TM uses several tapes, but it could be converted to a one-tape TM using the construction of Section 8.4.1. The first tape represents the entire memory of the computer. We have used a code in which addresses of memory words, in numerical order, alternate with the contents of those memory words. Both addresses and contents are written in binary. The marker symbols * and # are used to make it easy to find the ends of addresses and contents, and to tell whether a binary string is an address or contents. Another marker, \$, indicates the beginning of the sequence of addresses and contents.

The second tape is the "instruction counter." This tape holds one integer in binary, which represents one of the memory locations on tape 1. The value stored in this location will be interpreted as the next computer instruction to be executed.

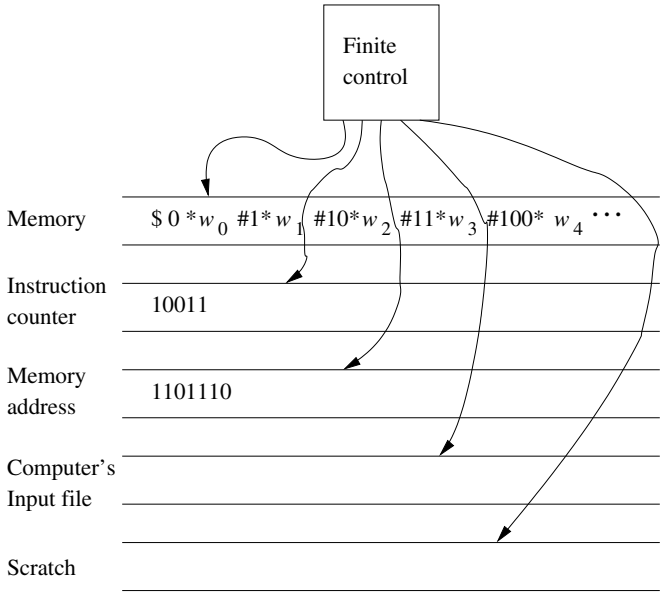

Figure 8.22: A Turing machine that simulates a typical computer

The third tape holds a "memory address" or the contents of that address after the address has been located on tape 1. To execute an instruction, the TM must find the contents of one or more memory addresses that hold data involved in the computation. First, the desired address is copied onto tape 3 and compared with the addresses on tape 1, until a match is found. The contents of this address is copied onto the third tape and moved to wherever it is needed, typically to one of the low-numbered addresses that represent the registers of the computer.

Our TM will simulate the *instruction cycle* of the computer, as follows.

1. Search the first tape for an address that matches the instruction number on tape 2. We start at the $ on the first tape, and move right, comparing each address with the contents of tape 2. The comparison of addresses on the two tapes is easy, since we need only move the tape heads right, in tandem, checking that the symbols scanned are always the same.

2. When the instruction address is found, examine its value. Let us assume that when a word is an instruction, its first few bits represent the action to be taken (e.g., copy, add, branch), and the remaining bits code an address or addresses that are involved in the action.

3. If the instruction requires the value of some address, then that address will be part of the instruction. Copy that address onto the third tape, and mark the position of the instruction, using a second track of the first tape

(not shown in Fig. 8.22), so we can find our way back to the instruction, if necessary. Now, search for the memory address on the first tape, and copy its value onto tape 3, the tape that holds the memory address.

4. Execute the instruction, or the part of the instruction involving this value. We cannot go into all the possible machine instructions. However, a sample of the kinds of things we might do with the new value are:

   (a) Copy it to some other address. We get the second address from the instruction, find this address by putting it on tape 3 and searching for the address on tape 1, as discussed previously. When we find the second address, we copy the value into the space reserved for the value of that address. If more space is needed for the new value, or the new value uses less space than the old value, change the available space by *shifting over*. That is:

      i. Copy, onto a scratch tape, the entire nonblank tape to the right of where the new value goes.

      ii. Write the new value, using the correct amount of space for that value.

      iii. Recopy the scratch tape onto tape 1, immediately to the right of the new value.

      As a special case, the address may not yet appear on the first tape, because it has not been used by the computer previously. In this case, we find the place on the first tape where it belongs, shift-over to make adequate room, and store both the address and the new value there.

   (b) Add the value just found to the value of some other address. Go back to the instruction to locate the other address. Find this address on tape 1. Perform a binary addition of the value of that address and the value stored on tape 3. By scanning the two values from their right ends, a TM can perform a ripple-carry addition with little difficulty. Should more space be needed for the result, use the shifting-over technique to create space on tape 1.

   (c) The instruction is a "jump," that is, a directive to take the next instruction from the address that is the value now stored on tape 3. Simply copy tape 3 to tape 2 and begin the instruction cycle again.

5. After performing the instruction, and determining that the instruction is not a jump, add 1 to the instruction counter on tape 2 and begin the instruction cycle again.

There are many other details of how the TM simulates a typical computer. We have suggested in Fig. 8.22 a fourth tape holding the simulated input to the computer, since the computer must read its input (the word whose membership in a language it is testing) from a file. The TM can read from this tape instead.

A scratch tape is also shown. Simulation of some computer instructions might make effective use of a scratch tape or tapes to compute arithmetic operations such as multiplication.

Finally, we assume that the computer makes an output that tells whether or not its input is accepted. To translate this action into terms that the Turing machine can execute, we shall suppose that there is an "accept" instruction of the computer, perhaps corresponding to a function call by the computer to put `yes` on an output file. When the TM simulates the execution of this computer instruction, it enters an accepting state of its own and halts.

While the above discussion is far from a complete, formal proof that a TM can simulate a typical computer, it should provide you with enough detail to convince you that a TM is a valid representation for what a computer can do. Thus, in the future, we shall use only the Turing machine as the formal representation of what can be computed by any kind of computing device.

8.6.3 Comparing the Running Times of Computers and Turing Machines

We now must address the issue of running time for the Turing machine that simulates a computer. As we have suggested previously:

- The issue of running time is important because we shall use the TM not only to examine the question of what can be computed at all, but what can be computed with enough efficiency that a problem's computer-based solution can be used in practice.
- The dividing line separating the *tractable* — that which can be solved efficiently — from the *intractable* — problems that can be solved, but not fast enough for the solution to be usable — is generally held to be between what can be computed in polynomial time and what requires more than any polynomial running time.
- Thus, we need to assure ourselves that if a problem can be solved in polynomial time on a typical computer, then it can be solved in polynomial time by a Turing machine, and conversely. Because of this polynomial equivalence, our conclusions about what a Turing machine can or cannot do with adequate efficiency apply equally well to a computer.

Recall that in Section 8.4.3 we determined that the difference in running time between one-tape and multitape TM's was polynomial — quadratic, in particular. Thus, it is sufficient to show that anything the computer can do, the multitape TM described in Section 8.6.2 can do in an amount of time that is polynomial in the amount of time the computer takes. We then know that the same holds for a one-tape TM.

Before giving the proof that the Turing machine described above can simulate n steps of a computer in $O(n^3)$ time, we need to confront the issue of

multiplication as a computer instruction. The problem is that we have not put a limit on the number of bits that one computer word can hold. If, say, the computer were to start with a word holding integer 2, and were to multiply that word by itself for n consecutive steps, then the word would hold the number 2^{2^n}. This number requires $2^n + 1$ bits to represent, so the time the Turing machine takes to simulate these n instructions would be exponential in n, at least.

One approach is to insist that words retain a fixed maximum length, say 64 bits. Then, multiplications (or other operations) that produced a word too long would cause the computer to halt, and the Turing machine would not have to simulate it any further. We shall take a more liberal stance: the computer may use words that grow to any length, but one computer instruction can only produce a word that is one bit longer than the longer of its arguments.

Example 8.16: Under the above restriction, addition is allowed, since the result can only be one bit longer than the maximum length of the addends. Multiplication is not allowed, since two m-bit words can have a product of length $2m$. However, we can simulate a multiplication of m-bit integers by a sequence of m additions, interspersed with shifts of the multiplicand one bit left (which is another operation that only increases the length of the word by 1). Thus, we can still multiply arbitrarily long words, but the time taken by the computer is proportional to the square of the length of the operands. □

Assuming one-bit maximum growth per computer instruction executed, we can prove our polynomial relationship between the two running times. The idea of the proof is to notice that after n instructions have been executed, the number of words mentioned on the memory tape of the TM is $O(n)$, and each computer word requires $O(n)$ Turing-machine cells to represent it. Thus, the tape is $O(n^2)$ cells long, and the TM can locate the finite number of words needed by one computer instruction in $O(n^2)$ time.

There is, however, one additional requirement that must be placed on the instructions. Even if the instruction does not produce a long word as a result, it could take a great deal of time to compute the result. We therefore make the additional assumption that the instruction itself, applied to words of length up to k, can be performed in $O(k^2)$ steps by a multitape Turing machine. Surely the typical computer operations, such as addition, shifting, and comparison of values, can be done in $O(k)$ steps of a multitape TM, so we are being overly liberal in what we allow a computer to do in one instruction.

Theorem 8.17: If a computer:

1. Has only instructions that increase the maximum word length by at most 1, and

2. Has only instructions that a multitape TM can perform on words of length k in $O(k^2)$ steps or less,

then the Turing machine described in Section 8.6.2 can simulate n steps of the computer in $O(n^3)$ of its own steps.

PROOF: Begin by noticing that the first (memory) tape of the TM in Fig. 8.22 starts with only the computer's program. That program may be long, but it is fixed and of constant length, independent of n, the number of instruction steps the computer executes. Thus, there is some constant c that is the largest of the computer's words and addresses appearing in the program. There is also a constant d that is the number of words occupied by the program.

Thus, after executing n steps, the computer cannot have created any words longer than $c + n$, and therefore, it cannot have created or used any addresses that are longer than $c+n$ bits either. Each instruction creates at most one new address that gets a value, so the total number of addresses after n instructions have been executed is at most $d + n$. Since each address-word combination requires at most $2(c+n)+2$ bits, including the address, the contents, and two marker symbols to separate them, the total number of TM tape cells occupied after n instructions have been simulated is at most $2(d + n)(c + n + 1)$. As c and d are constants, this number of cells is $O(n^2)$.

We now know that each of the fixed number of lookups of addresses involved in one computer instruction can be done in $O(n^2)$ time. Since words are $O(n)$ in length, our second assumption tells us that the instructions themselves can each be carried out by a TM in $O(n^2)$ time. The only significant, remaining cost of an instruction is the time it takes the TM to create more space on its tape to hold a new or expanded word. However, shifting-over involves copying at most $O(n^2)$ data from tape 1 to the scratch tape and back again. Thus, shifting-over also requires only $O(n^2)$ time per computer instruction.

We conclude that the TM simulates one step of the computer in $O(n^2)$ of its own steps. Thus, as we claimed in the theorem statement, n steps of the computer can be simulated in $O(n^3)$ steps of the Turing machine. □

As a final observation, we now see that cubing the number of steps lets a multitape TM simulate a computer. We also know from Section 8.4.3 that a one-tape TM can simulate a multitape TM by squaring the number of steps, at most. Thus:

Theorem 8.18: A computer of the type described in Theorem 8.17 can be simulated for n steps by a one-tape Turing machine, using at most $O(n^6)$ steps of the Turing machine. □

8.7 Summary of Chapter 8

- ✦ *The Turing Machine*: The TM is an abstract computing machine with the power of both real computers and of other mathematical definitions of what can be computed. The TM consists of a finite-state control and an infinite tape divided into cells. Each cell holds one of a finite number

of tape symbols, and one cell is the current position of the tape head. The TM makes moves based on its current state and the tape symbol at the cell scanned by the tape head. In one move, it changes state, overwrites the scanned cell with some tape symbol, and moves the head one cell left or right.

- *Acceptance by a Turing Machine*: The TM starts with its input, a finite-length string of tape symbols, on its tape, and the rest of the tape containing the blank symbol on each cell. The blank is one of the tape symbols, and the input is chosen from a subset of the tape symbols, not including blank, called the input symbols. The TM accepts its input if it ever enters an accepting state.

- *Recursively Enumerable Languages*: The languages accepted by TM's are called recursively enumerable (RE) languages. Thus, the RE languages are those languages that can be recognized or accepted by any sort of computing device.

- *Instantaneous Descriptions of a TM*: We can describe the current configuration of a TM by a finite-length string that includes all the tape cells from the leftmost to the rightmost nonblank. The state and the position of the head are shown by placing the state within the sequence of tape symbols, just to the left of the cell scanned.

- *Storage in the Finite Control*: Sometimes, it helps to design a TM for a particular language if we imagine that the state has two or more components. One component is the control component, and functions as a state normally does. The other components hold data that the TM needs to remember.

- *Multiple Tracks*: It also helps frequently if we think of the tape symbols as vectors with a fixed number of components. We may visualize each component as a separate track of the tape.

- *Multitape Turing Machines*: An extended TM model has some fixed number of tapes greater than one. A move of this TM is based on the state and on the vector of symbols scanned by the head on each of the tapes. In a move, the multitape TM changes state, overwrites symbols on the cells scanned by each of its tape heads, and moves any or all of its tape heads one cell in either direction. Although able to recognize certain languages faster than the conventional one-tape TM, the multitape TM cannot recognize any language that is not RE.

- *Nondeterministic Turing Machines*: The NTM has a finite number of choices of next move (state, new symbol, and head move) for each state and symbol scanned. It accepts an input if any sequence of choices leads to an ID with an accepting state. Although seemingly more powerful than

the deterministic TM, the NTM is not able to recognize any language that is not RE.

- *Semi-infinite-Tape Turing Machines*: We can restrict a TM to have a tape that is infinite only to the right, with no cells to the left of the initial head position. Such a TM can accept any RE language.

- *Multistack Machines*: We can restrict the tapes of a multitape TM to behave like a stack. The input is on a separate tape, which is read once from left-to-right, mimicking the input mode for a finite automaton or PDA. A one-stack machine is really a DPDA, while a machine with two stacks can accept any RE language.

- *Counter Machines*: We may further restrict the stacks of a multistack machine to have only one symbol other than a bottom-marker. Thus, each stack functions as a counter, allowing us to store a nonnegative integer, and to test whether the integer stored is 0, but nothing more. A machine with two counters is sufficient to accept any RE language.

- *Simulating a Turing Machine by a real computer*: It is possible, in principle, to simulate a TM by a real computer if we accept that there is a potentially infinite supply of a removable storage device such as a disk, to simulate the nonblank portion of the TM tape. Since the physical resources to make disks are not infinite, this argument is questionable. However, since the limits on how much storage exists in the universe are unknown and undoubtedly vast, the assumption of an infinite resource, as in the TM tape, is realistic in practice and generally accepted.

- *Simulating a Computer by a Turing Machine*: A TM can simulate the storage and control of a real computer by using one tape to store all the locations and their contents: registers, main memory, disks, and other storage devices. Thus, we can be confident that something not doable by a TM cannot be done by a real computer.

8.8 Gradiance Problems for Chapter 8

The following is a sample of problems that are available on-line through the Gradiance system at `www.gradiance.com/pearson`. Each of these problems is worked like conventional homework. The Gradiance system gives you four choices that sample your knowledge of the solution. If you make the wrong choice, you are given a hint or advice and encouraged to try the same problem again.

Problem 8.1: A nondeterministic Turing machine M with start state q_0 and accepting state q_f has the following transition function:

$\delta(q,a)$	0	1	B
q_0	$\{(q_1,0,R)\}$	$\{(q_1,0,R)\}$	$\{(q_1,0,R)\}$
q_1	$\{(q_1,1,R),(q_2,0,L)\}$	$\{(q_1,1,R),(q_2,1,L)\}$	$\{(q_1,1,R),(q_2,B,L)\}$
q_2	$\{(q_f,0,R)\}$	$\{(q_2,1,L)\}$	$\{\}$
q_f	$\{\}$	$\{\}$	$\{\}$

Deduce what M does on any input of 0's and 1's. Demonstrate your understanding by identifying, from the list below, the ID that **cannot** be reached on some number of moves from the initial ID X [shown on-line by the Gradiance system].

Problem 8.2: For the Turing machine in Problem 8.1, simulate all sequences of 5 moves, starting from initial ID $q_0$1010. Find, in the list below, one of the ID's reachable from the initial ID in **exactly** 5 moves.

Problem 8.3: The Turing machine M has:

1. States q and p; q is the start state.
2. Tape symbols 0, 1, and B; 0 and 1 are input symbols, and B is the blank.
3. The next-move function in Fig. 8.23.

Your problem is to describe the property of an input string that makes M halt. Identify a string that makes M halt from the list below.

State	Tape Symbol	Move
q	0	$(q,0,R)$
q	1	$(p,0,R)$
q	B	(q,B,R)
p	0	$(q,0,L)$
p	1	none (halt)
p	B	$(q,0,L)$

Figure 8.23: A Turing machine

Problem 8.4: Simulate the Turing machine M of Fig. 8.23 on the input 1010110, and identify one of the ID's (instantaneous descriptions) of M from the list below.

Problem 8.5: A Turing machine M with start state q_0 and accepting state q_f has the following transition function:

$\delta(q,a)$	0	1	B
q_0	$(q_0,1,R)$	$(q_1,1,R)$	(q_f,B,R)
q_1	$(q_2,0,L)$	$(q_2,1,L)$	(q_2,B,L)
q_2	–	$(q_0,0,R)$	–
q_f	–	–	–

Deduce what M does on any input of 0's and 1's. Hint: consider what happens when M is started in state q_0 at the left end of a sequence of any number of 0's (including zero of them) and a 1. Demonstrate your understanding by identifying the true transition of M from the list below.

8.9 References for Chapter 8

The Turing machine is taken from [8]. At about the same time there were several less machine-like proposals for characterizing what can be computed, including the work of Church [1], Kleene [5], and Post [7]. All these were preceded by the work of Gödel [3], which in effect showed that there was no way for a computer to answer all mathematical questions.

The study of multitape Turing machines, especially the matter of how their running time compares with that of the one-tape model initiated with Hartmanis and Stearns [4]. The examination of multistack and counter machines comes from [6], although the construction given here is from [2].

The approach in Section 8.1 of using "hello, world" as a surrogate for acceptance or halting by a Turing machine appeared in unpublished notes of S. Rudich.

1. A. Church, "An undecidable problem in elementary number theory," *American J. Math.* **58** (1936), pp. 345–363.

2. P. C. Fischer, "Turing machines with restricted memory access," *Information and Control* **9**:4 (1966), pp. 364–379.

3. K. Gödel, "Über formal unentscheidbare Sätze der Principia Mathematica und verwander Systeme," *Monatschefte für Mathematik und Physik* **38** (1931), pp. 173–198.

4. J. Hartmanis and R. E. Stearns, "On the computational complexity of algorithms," *Transactions of the AMS* **117** (1965), pp. 285–306.

5. S. C. Kleene, "General recursive functions of natural numbers," *Mathematische Annalen* **112** (1936), pp. 727–742.

6. M. L. Minsky, "Recursive unsolvability of Post's problem of 'tag' and other topics in the theory of Turing machines," *Annals of Mathematics* **74**:3 (1961), pp. 437–455.

7. E. Post, "Finite combinatory processes-formulation," *J. Symbolic Logic* **1** (1936), pp. 103–105.

8. A. M. Turing, "On computable numbers with an application to the Entscheidungsproblem," *Proc. London Math. Society* **2**:42 (1936), pp. 230–265. See also *ibid.* **2**:43, pp. 544–546.