Chapter

12

Models of Computation

Main content

Chapter Contents

Chapter Introduction

r studying this chapter, you will be able to:

Explain the purpose of constructing a model
List the required features of a computing agent

Describe the components of a Turing machine, and explain how it is a good model of a computing agent
List the features of an algorithm, and explain how a Turing machine program matches them
Simulate the operation of a simple Turing machine on specific inputs
Construct a simple Turing machine from a specification, both writing the rules and drawing the state diagram
State the Church-Turing thesis and explain what it means
Justify why computer scientists believe the Church-Turing thesis is true
Explain what an unsolvable problem is
Describe the halting problem, what its inputs and outputs are

Outline the proof that the halting problem cannot be solved by any Turing machine

Main content

Chapter Contents

12.1 Introduction

The central topic of this book has been, in one way or another, algorithmic problem solving. We've discussed the concept of an algorithm, how to represent algorithms, their correctness and efficiency, the hardware that executes algorithms, the levels of abstraction in which a programmer deals with algorithms, and, finally, the system software that translates these abstractions back to the elementary hardware level. It might seem as though algorithms, and the problems solvable by algorithms, represent the entire scope of the computer science universe.

However, one of the most fundamental ideas in mathematics and computer science is *that there are problems that do not have any algorithmic solution*! Be sure you understand why this is such a powerful statement. There are many problems for which no algorithmic solution has yet been found, but for which we might find one if we were clever enough to discover it. Indeed, such new discoveries are being made all the time. But there are also problems for which no algorithmic solution exists; it does not matter how inventive we may be, how much time we spend looking, or how remarkable our hardware or software; *no algorithms will ever be found that solve these problems*.

We will prove this statement later in this chapter by actually finding such a problem. This is a difficult task because failing to find an algorithmic solution to a problem does not prove that one does not exist. It might only mean that we have not yet been able to discover such an algorithm. Instead, we must show that no one can ever find such an algorithm—that one does not exist.

Algorithms, as noted in Chapter 1, are carried out by computing agents (such as people, robots, and computers). Throughout most of this book, we've assumed that the computing agent is a real computer. Ordinarily, we would choose to execute an algorithm on the most modern, high-speed computer available, with all the bells and whistles we could possibly find. But to show that something cannot be done by *any* computer, we want the bells and whistles to get out of the way so we can concentrate on the fundamental nature of "computerness." What we need is a simple, "idealized" computer—something easy to work with yet theoretically as powerful as the real thing. We need a *model* of a computer; indeed, to consider algorithms in general, we need a model of a computing agent.

12.2 What Is a Model?

Model cars, model trucks, model airplanes, and dolls (model people) are forever popular with children. Children use these toys to "play" at being grown up—at being drivers, pilots, and parents—because the toys capture the spirit of the objects they model. A model car looks like a car. The more expensive the model, the more realistic its features. But although the model captures the essence of a car, it is smaller in scale, omits many of the details of a real car, and does not have the full functionality of a real car.

Models are also an important way of studying many physical and social phenomena. Weather systems, climate cycles, the spread of epidemics, population demographics, and chemical molecules—all are phenomena that have been studied via modeling. (In fact, we will look at some of these applications in Chapter 13.) Like a model car, a model of such a phenomenon

1. Captures the essence—the important properties—of the real thing;
2. Probably differs in scale from the real thing;
3. Omits some of the details of the real thing; and
4. Lacks the full functionality of the real thing.

The model might be a physical model or a pencil-and-paper mathematical model. For example, a physical model of a chemical molecule might use Velcro®-covered balls stuck together in a certain way to represent the molecular structure. This model illustrates certain important properties: how many atoms of each element are present and where they are located in relation to one another. It is much larger than the real molecule, does not display the details of the chemical bonding, and is certainly not a real molecule. An alternative "physical" model—a computer visualization—is shown in Chapter 13, Figure 13.12.

A simple example of a mathematical model is the equation for the distance d that a moving vehicle travels as the product of rate *r* and time *t*:

A calculation that a vehicle traveling at a constant rate of 60 miles per hour for 2 hours will cover a distance of 120 miles can be done in an instant by simply plugging values into the equation. But this equation is a simplification of reality because it assumes that the rate, that is, the speed of the vehicle, is constant throughout the entire two-hour period, so this result is only an approximation. Because this is not a physical model, it does not have a size as such, but there is a difference in time scale from the actual moving vehicle.

What can be gained by studying models if they do not behave in exactly the same way as the real thing? For one thing, they can enhance our understanding of the real system being modeled. By changing some aspect within the model, we can immediately see the effects of that change. These changes might be very costly, difficult, or dangerous to make in the real phenomena. The benefit is that models give us a safe and controlled environment to play with "what ifs"—what might be the effect if this or that factor in the real system were changed? The answers can be used to guide future decisions. Models can also provide environments for learning and practicing interactions with various

phenomena. An aircraft flight simulator, for example, can give the trainee pilot realistic experience in a danger-free setting. Finally, not only can models give us information about existing phenomena, they also can be used as design tools. A model of a new design may reveal major flaws without the time, expense, and potential danger of building a prototype. (We will look more closely at these applications of models in Chapter 13.)

Whether a model is used to predict the behavior of an existing system or as a test bed for a proposed design, the information it provides is only as good as the assumptions made in building the model. If the model does not incorporate the major aspects of the system being studied, if relationships are represented incorrectly, or if so much detail has been omitted as to make the model a totally inaccurate representation, then little faith can be placed in the results it produces.

Main content

Chapter Contents

12.3 A Model of a Computing Agent

12.3.1 Properties of a Computing Agent

To construct a good model of the "computing agent" entity—one that enables us to explore the capabilities and limitations of computation in the most general sense—we must make certain that we capture the fundamental properties of a computing agent while suppressing lower-level details. This means we must decide which features are central to a computing agent and which are relatively incidental and can be ignored. For example, a computing agent must be able to follow the instructions in an algorithm. The instructions must be presented in some form that the computing agent can comprehend, but it does not matter whether the instructions are presented in English or Japanese, as binary strings, or even as pictures.

Likewise, the computing agent must be able to receive any data pertinent to the task. When we dealt with real computers, we described this as an input task, but the ability to accept input is central to any computing agent—whether a human being or a programmable thermostat. The instructions and data must be stored somewhere during the execution of the algorithm. In addition, they must be retrievable, whether from a computer's memory, the thermostat microprocessor's memory, a human being's memory, or a written sheet of paper.

The computing agent must be able to act in accordance with algorithm instructions. These instructions may take into account the present situation or state of the computing agent, as well as the particular input item being processed. In a real computer, a conditional operation may say, "If condition A then do B else do C." Condition A may involve the value of some variable or variables that have already been read into memory; we may think of the contents of memory (i.e., how the various bits are set) as the present state of the computer. The thermostat microprocessor may have an instruction that says, "If the temperature is greater than 74 degrees and I am programmed to keep the temperature at or below 74 degrees, then turn the cooling control to ON." Here the action of the thermostat depends on both the input of the

current temperature from its temperature sensor and the "state" of its programming, just as a human being carrying out the algorithm of ordering lunch from a menu reacts both to the "input" (what items are on the menu) and to his or her present state of hunger.

Finally, the computing agent is expected to produce output because the outcome of an algorithm must be an observable result. The computer displays results on a screen, prints them on a sheet of paper, or writes them to a file; the thermostat displays the ON signal for the cooling system; the human being speaks or writes.

Practice Problems

The mathematical model of the relationship between distance, rate, and time gives a better approximation of reality if it is applied over smaller time segments, assuming a constant rate within each time segment. Use the following data to approximate the distance traveled over a two-hour period.

Time period	Rate during that period
1:00–1:30 p.m.	58 mph
1:30–2:00 p.m.	61 mph
2:00–2:30 p.m.	57 mph
2:30–3:00 p.m.	62 mph

2. Answer

Describe a situation (besides aircraft pilot training) in which a simulator would be useful as a training device.

Answer

Piloting a boat, performing an operation, fighting a fire

What factors might a model of groundwater pollution need to include? What are the advantages of a good model? Are there potential disadvantages to using such a model?

Answer

Soil conditions, water supply, types of industrial waste. It could illustrate long-term effects of various waste-disposal policies. If it were inaccurate, policies based on the model might be pursued that would result in environmental damage.

To summarize, we require that any computing agent be able to do all of the following:

1. Accept input.
2. Store information in memory and retrieve it from memory.

3. Take actions according to algorithm instructions; the choice of what action to take may depend on the present state of the computing agent, as well as on the input item presently being processed.
4. Produce output.

Of course, a real computer has all of these capabilities and is an example of a computing agent, as are a human being and a programmable thermostat. The thermostat, however, has a very limited set of primitive operations it can perform, so it can react only to a very limited algorithm. The computer, although it has a limited set of simple primitives, is a general-purpose computing agent because, as we have seen in the previous chapters, those primitives can be combined and organized to accomplish complex tasks.

The "primitive operations" available to human beings haven't been fully explored, but in many ways they seem to exceed those of a computer, and we would certainly classify a human being as a general-purpose computing agent.

In the next section, we will discuss one particular model for a computing agent. It will have the four required properties just specified, and it will represent a general-purpose computing agent able to follow the instructions of many different algorithms.

Main content

Chapter Contents

12.3.2 The Turing Machine

We think of "computing" as a modern activity–something done by electroniccomputers. But interest in the theoretical nature of computation far predates the advent of modern computers. By the end of the 19th century, mathematicians were interested in formalizing the nature of proof, with two goals in mind. First, a formal basis for mathematical proofs would guarantee the correctness of a proof because the proof would contain no intuitive statements, such as "It is clear that . . ." or "We can now see that . . . ." Second, a formal basis for proofs might allow for mechanical theorem-proving, where correct proofs could be generated simply by following a set of rules. In 1931, the Austrian logician Kurt Gödel looked at formal systems to describe the ordinary arithmetic of numbers. He demonstrated that in any reasonable system, there are true statements about arithmetic that cannot be proved using that system. This led to interest in finding a way to recognize which statements are indeed unprovable in a formal system—that is, in finding a computational procedure (what we have called an algorithm) to recognize such statements. This in turn led to an investigation of the nature of computation itself, and a number of mathematicians in the mid-1930s proposed various models of computational procedures, along with models of computing agents to carry out those procedures. We will look at the model proposed by Alan Turing.

Alan Turing, Brilliant Eccentric

Alan Turing (1912–1954) was a brilliant British mathematician and a colorful individual. Stories abound about his "absentminded professor" demeanor, his interest in running (through the streets of London with an alarm clock flopping about, tied to his belt by a piece of twine), and his fascination with a children's radio show whose characters he would discuss daily with his mother. Convicted of homosexual acts in 1952, he chose drug treatment over prison, primarily because he feared a prison term would

impede his intellectual work. There was a Broadway play (*Breaking the Code*) written about him, years after his death by suicide. In 2014, a film about Turing's life, *The Imitation Game* starring British actor Benedict Cumberbatch, was released. It was the highest-grossing independent film of 2014, and in 2015 was nominated for eight Academy Awards and nine British Academy of Film and Television Arts awards.

Turing made three distinct and remarkable contributions to computer science. First, he devised what is now known as the Turing machine, using it—as we will see in this chapter—to model computation and to discover that some problems have no general computable solution. Second, during World War II, his team at the British Foreign Office built the Colossus machine, which used cryptanalysis, the science of code breaking, to break the secret code used on the German Enigma machine. The details of this work, carried out in a Victorian country mansion called Bletchley Park, were kept secret until many years later. Breaking the code enabled the British to gain access to intelligence about German military movements, which contributed significantly toward winning the war. Third, after the war Turing investigated what it means for machines to "think." We'll discuss his early contribution to *artificial intelligence* in Chapter 15.

Starting in 2011, a restoration project began on the by-then decrepit buildings of Bletchley Park. Because of the secrecy surrounding the work there during the war, there were no photographs of the building interiors. The restoration team contacted some veterans of Bletchley Park to help supply authentic details. In June 2014, the restored Bletchley Park was opened to the public by the Duchess of Cambridge, wife of Britain's Prince William, who walked through one of the restored code breaking "huts" where her paternal grandmother had worked during World War II and talked with a Bletchley Park veteran who had been a colleague of her grandmother.

A **Turing machine** is a theoretical model of computation that includes a (conceptual) tape extending infinitely in both directions. The tape is divided into cells, each of which contains one symbol. The symbols must come from a finite set of symbols called the **tape alphabet**. The tape alphabet for a given Turing machine always contains a special symbol b (for "blank"), usually both of the symbols 0 and 1 (zero and one), and sometimes a limited number of other symbols, let's say X and Y, used as placeholders or markers of some kind. At any point in time, only a finite number of the cells contain nonblank symbols. Figure 12.1 shows a typical tape configuration, with three nonblank cells containing the alphabet symbols 0, 1, 1, respectively.

Figure 12.1 A Turing machine tape

The tape is used to hold the input to the Turing machine. We know that input must be presented to a computing agent in a form it can understand; for a Turing machine, this means that the input must be expressed as a finite string of nonblank symbols from the tape alphabet. The Turing machine writes its output on the tape, again using the same alphabet of symbols. The tape also serves as memory.

The rest of the Turing machine consists of a unit that reads one cell of the tape at a time and writes a symbol in that cell. There is a finite number k of "states" of the machine, labeled 1, 2, … , k, and at any moment the unit is in one of these states. A state can be thought of as a certain condition; the Turing machine may reach this condition partly on the basis of its history of events, much as your "hungry state" is a condition reached because of the meals you have skipped recently.

Figure 12.2 shows a particular Turing machine configuration. Using the tape of Figure 12.1, the machine is currently in state 1 and is reading the cell containing the symbol 0, so the 0 is what the machine is seeing as the current input symbol.

Figure 12.2A Turing machine configuration

The Turing machine is designed to carry out only one type of primitive operation. Each time such an operation is done, three actions take place:

1. Write a symbol in the cell (replacing the symbol already there).
2. Go into a new state (it might be the same as the current state).
3. Move the "read head" one cell left or right.

The details of the actions (what to write, what the new state is, and which direction to move) depend on the current state of the machine and on the contents of the tape cell currently being read (the input). **Turing machine instructions** describe these details. Each instruction tells what to do for a specific current state and current input symbol, as follows:

The Turing machine's single primitive operation is to check its current state and the current input symbol being read, look for an instruction that tells what to do under these circumstances, and then carry out the three actions specified by that instruction. For example, one Turing machine instruction might say

If a Turing machine is in the configuration shown in Figure 12.2 (where the current state is 1 and the current input symbol is 0), then this instruction applies. After the machine executes this instruction, its next configuration is shown in Figure 12.3, where the previous 0 symbol has been overwritten with a 1, the state has changed to state 2, and the read head has moved one cell to the right on the tape.

Figure 12.3The next Turing machine configuration after executing one instruction

Let's develop a shorthand notation for Turing machine instructions. There are five components:

- Current state
- Current symbol
- Next symbol
- Next state
- Direction of move

We'll write these five things in that order, enclosed in parentheses.

The instruction that we talked about earlier,

is therefore represented by the 5-tuple:

(1,0,1,2,*R*)

Similarly, the Turing machine instruction

(2,1,1,2,*L*)

stands for

In following this instruction, the machine writes in the current cell the same symbol (1) as was already there and remains in the same state (state 2) as before. It also moves one cell to the left.

A Turing machine can execute a whole sequence of instructions. A clock governs the action of the machine. Whenever the clock ticks, the Turing machine performs its primitive operation; that is, it looks for an instruction that applies to its current state and the symbol currently being read and then follows that instruction. Instructions may be used more than once.

We have glossed over a couple of important details. What if there is more than one instruction that applies to the current configuration? Suppose, as in Figure 12.2, that the current state is 1, that the current symbol is 0, and that

(1,0,1,2,*R*)

(1,0,0,3,*L*)

both appear in the same collection of instructions. These instructions are in conflict. Should the Turing machine write a 1, go to state 2, and move right, or should it write a 0, go to state 3, and move left? We can eliminate this ambiguity by requiring that a set of instructions for a Turing machine can never contain two or more instructions that begin with the same two values, as in the following:

(*i, j,* -, -, -)

(*i, j,* -, -, -)

On the other hand, what if there is no instruction that applies to the current state and current symbol for the machine? That is, we are currently in state a reading symbol b, but there is no instruction of the form (*a*,*b*,-,-,-). In this case, we specify that the machine halts, doing nothing further. This is the stopping condition for a Turing machine.

We impose two additional conventions on the Turing machine regarding its initial configuration when the clock begins. The start-up state is always state 1, and the machine is always reading the leftmost nonblank cell on the tape. This ensures that the Turing machine has a fixed and definite starting point.

Now let's do a sample Turing machine computation. Suppose the instructions available to a Turing machine are

1. (1,0,1,2,*R*)
2. (1,1,1,2,*R*)
3. (2,0,1,2,*R*)
4. (2,1,0,2,*R*)

5. (2,*b*,*b*,3,*L*)
   Also suppose the Turing machine's initial configuration is again that of Figure 12.2, reprinted here:

   This satisfies our convention about starting in state 1 at the leftmost nonblank cell on the tape. The Turing machine looks for an appropriate instruction for its current state, 1, and its current input symbol, 0, which means it looks for an instruction of the form (1,0,-,-,-). Instruction 1 applies; this was our sample instruction earlier, and the resulting configuration agrees with Figure 12.3:

   At the next clock tick, with current state 2 and current symbol 1, the Turing machine looks for an instruction of the form (2,1,-,-,-). Instruction 4 applies and, after the appropriate actions are performed, the resulting configuration is

   Instruction 4 applies again and results in

   Instruction 5 now applies, leading to

   At this point, the machine is in state 3 reading the symbol 0. Because there are no instructions of the form (3,0,-,-,-), the machine halts. The Turing machine computation is complete.

   Although we numbered this collection of instructions for reference, the Turing machine does not necessarily execute instructions in the order of this numbering. Some instructions may not be executed at all, and some may be executed more than once. The sequence of instructions used depends on the input written on the tape.

   As we have seen, the Turing machine (despite its name) is not a machine at all. It is a theoretical model of the pencil-and-paper type designed to capture the essential features of a computing agent. So, how well does the Turing machine stack up against our list of required features for a computing agent?

1. *It can accept input*—The Turing machine can read symbols on its tape.
2. *It can store information in memory and retrieve it from memory*—The Turing machine can write symbols on its tape and, by moving around over the tape, can go back and read those symbols at a later time. The tape serves as the Turing machine memory.
3. *It can take actions according to algorithm instructions, and the choice of action to take may depend on the present state of the computing agent and on the input item presently being processed*—Certainly the Turing machine satisfies this requirement insofar as Turing machine instructions are concerned; the present state and present symbol being processed determine the appropriate instruction, and that instruction specifies the actions to be taken.

4. *It can produce output*—The Turing machine writes symbols on its tape in the course of its normal operation. If (when?) the Turing machine halts, what is written on the tape at that time can be considered output.
   In the Turing machine computation that we just finished, the input was the string of symbols 011 (ignoring the surrounding blanks) and the output was the string of symbols 100. Starting with the same input tape but with a different set of instructions could result in different output; similarly, starting with the same instructions but with a different input tape could result in different output. Given the benefit of hindsight, we could say that we wrote this particular set of instructions to carry out the task of transforming the string 011 into the string 100. Writing a set of Turing machine instructions to allow a Turing machine to carry out a certain task is similar to writing a computer program in a programming language, such as those discussed in Chapters 9 and 10, to allow a real computer to carry out a certain task. We call such a collection of instructions a **Turing machine program**.

   Thus, a Turing machine does capture those properties we identified as essential for a computing agent, which qualifies it as a model of a computing agent. Furthermore, it represents a general computing agent in the sense that, like a real computer, it can follow many different sets of instructions (programs) and thus do many different things (unlike the one-jobonly thermostat). By its very simplicity of operation, it has eliminated many real-world details, such as exactly how symbols are read from or written to the tape, exactly how data are to be encoded into a string of symbols from the alphabet to be written on the tape, exactly how a string of symbols on the tape is to be interpreted as meaningful output, and exactly how the machine carries out the activities of "changing state." In fact, the Turing machine is such a simple concept that we may wonder how good a model it really is. Did we eliminate too many details? We'll answer the question of how good a model the Turing machine is later in the chapter.

   A Turing machine is different in scale from any real computing agent in one respect. A Turing machine can, given the appropriate instructions, move right or left to the blank portion of the tape and write a nonblank symbol. When this happens, the machine has gobbled up an extra cell to use for information storage purposes—that is, as memory. Depending on the instructions, this could happen over and over, which means that there is *no limit* to the amount of memory available to the machine. Any real computing agent has a limit on the memory available to it. In particular, a real computer, though it has a certain amount of internal memory and has access to external memory in the form of disks, tapes, or online storage, still has such a limit.

   This difference in scale means that a Turing machine (elementary device though it may seem to be) actually has more capability in one respect than any real computer that exists or ever will exist. Therefore, we must be careful about the use of the Turing machine model and the conclusions we draw from it about "real" computing (i.e., computing on a real computer). If we find some task that a Turing machine can perform (because of its limitless memory), it *might* not be a task that a real computer could perform.

Practice Problems

Given the Turing machine instruction (2,*b*,1,3,*L*) and the configuration

draw the next configuration.

Answer

A Turing machine has the following instructions:

(1,0,0,2,*R*)

(2,1,1,2,*L*)

(2,0,1,2,*R*)

(1,*b*,1,1,*L*)

For each of the following configurations of this Turing machine, draw the next configuration.

1.

Answer

2.

Answer

3.

Answer

4.

Answer

Consider a Turing machine that has the following two instructions:

(1,1,0,2,*R*)

(2,1,1,1,*R*)

Determine its output when it is run on the following tape. (Remember that a Turing machine starts in state 1, reading the leftmost nonblank cell.)

Answer

b 0 1 0 *b*

Using the Turing machine from Practice Problem 3, determine its output when it is run on the following tape.

Answer

b 0 0 0 *b*

12.4 A Model of an Algorithm

An algorithm is a collection of instructions intended for a computing agent to follow. If we accept the Turing machine as a model of a computing agent, then the instructions for a Turing machine should be a model of an algorithm. Remember from our definition in Chapter 1 that an algorithm must

1. Be a well-ordered collection;
2. Consist of unambiguous and effectively computable operations;
3. Halt in a finite amount of time; and
4. Produce a result.

Let's consider an arbitrary collection of Turing machine instructions and see whether it exhibits these properties of an algorithm.

1. *Be a well-ordered collection*—The Turing machine must know which operation to carry out first and which to do next at any step. We have already specified the initial conditions for a Turing machine computation: that the Turing machine must begin in state 1, reading the leftmost nonblank cell on the tape. We have also insisted that in any collection of Turing machine instructions, there cannot be two different instructions that both begin with the same current state and current symbol. Given this requirement, there is never any confusion about which operation to do next. There is *at most* one instruction that matches the current state and current symbol of the Turing machine. If there is one instruction, the Turing machine executes the operation that instruction describes. If there is no instruction, the Turing machine halts.
2. *Consist of unambiguous and effectively computable operations*—Recall that this property is *relative* to the computing agent; that is, operations must be understandable and doable by the computing agent. Each individual Turing machine instruction describes an operation that (to the Turing machine) is unambiguous, requiring no additional explanation, and any Turing machine is able to carry out the operation described. After all, Turing machine instructions were explicitly designed for Turing machines to execute.
3. *Halt in a finite amount of time*—For a Turing machine to halt when executing a collection of instructions, it must reach a configuration where no appropriate instruction exists. This depends on the input given to the Turing machine—that is, the contents initially written on the tape. Consider the following set of Turing machine instructions:

   (1,0,0,1,*R*)

   (1,*b*,*b*,1,*R*)

   and suppose the tape initially contains, as its nonblank portion, the single symbol 1. The initial configuration is

   and the machine halts immediately because there is no applicable instruction. On the other hand, suppose the same set of instructions is used with a starting tape that contains the single symbol 0. The Turing machine computation is then

   We can see that the second instruction applies indefinitely and that this Turing machine will never halt.

Typically, an algorithm is designed to carry out a certain type of task. Let us agree that *for input appropriate to that task*, the instructions must be such that the Turing machine does indeed eventually halt. If the Turing machine is run on a tape containing data that are not appropriate input for the task of interest, it need not halt.

This may seem to be a change in our definition of an algorithm, but it simply confirms that there is always a "universe of discourse" connected with the problem we are trying to solve. For example, we can use a simple algorithm for dividing one positive integer by another positive integer using repeated subtraction until the result is negative.
Thus, can be computed using this algorithm as follows:
The quotient is 2 because two subtractions could be done before the result became negative. However, if we attempt to use this same approach to compute , we get
and so on.

The process would never halt because the result would never become negative. Yet, this approach is still an algorithm *for the problem of dividing two positive numbers* because it does produce the correct result and then halt when given input suitable for this problem.

4. *Produce a result*—We have already imposed the requirement that the Turing machine instructions must lead to a halting configuration when executed on input appropriate to the problem being solved. Whatever is written on the tape when the machine halts is the result.
A collection of Turing machine instructions that obeys the restrictions we have specified satisfies the properties required of an algorithm. Yet, it is not a "real" algorithm because it is not designed to be executed by a "real" computing agent. It is a model of an algorithm, designed to be executed by the model computing agent called a Turing machine.

   Most of the time, no distinction is made between a Turing machine as a computing agent and the instructions (algorithm) it carries out—a machine together with a set of instructions is called "a Turing machine" and is thought of as an algorithm. Thus, we say we are going to write a Turing machine to do a particular task, when we really mean that we are going to write a set of instructions—a Turing machine program, an algorithm—to do that task.

Main content

Chapter Contents

12.5 Turing Machine Examples

Because the Turing machine is such a simple device, it may seem nearly impossible to write a program for a Turing machine that carries out any interesting or significant task. In this section, we look at a few Turing machines that, though they do not accomplish anything earthshaking, should convince you that Turing machines can do some rather worthwhile things.

Main content

12.5.1 A Bit Inverter

Let's assume that the only nonblank portion of the input tape for a particular Turing machine consists of a string of bits (0s and 1s). Our first Turing machine moves along its tape from left to right inverting all of the bits—that is, changing 0s to 1s and 1s to 0s. (Recall that our sample Turing machine computation inverted the bits in the string 011, resulting in the string 100. Do you think that machine is a bit inverter? What if the leftmost nonblank symbol on the input tape is a 1?)

The Turing machine must begin in state 1 on the leftmost nonblank cell. Whatever the current symbol that is read, the machine must invert it by printing its opposite. Machine state 1 must, therefore, be a state in which 0s are changed to 1s and 1s are changed to 0s. This is exactly what we want to happen everywhere along the tape, so the machine never needs to go to another state; it can simply move right while remaining in state 1. When we come to the final blank, we want to halt. This can be accomplished by making sure that our program does not contain any instruction of the form

(1, b,–,–,–)

We have now described the Turing machine algorithm in words, but let's represent it more precisely. In the past, we've used pseudocode to describe algorithms. Here we'll use an alternative form of representation that corresponds more closely to Turing machine instructions. A **state diagram** is a visual representation of a Turing machine algorithm, where circles represent states, and arrows represent transitions from one state to another. Along each transition arrow, we show three things: the input symbol that caused the transition, the corresponding output symbol to be written, and the direction of movement. For the bit inverter Turing machine, we have only one state and hence one circle in the state diagram, shown in Figure 12.4.

Figure 12.4 State diagram for the bit inverter machine

The arrow originating in and returning to state 1 marked $1/0/R$ says that when in state 1 (the only state) reading an input symbol of 1, the machine should print the symbol 0, move right, and remain in state 1. The arrow marked $0/1/R$ says that when in state 1 reading an input symbol of 0, the machine should print the symbol 1, move right, and remain in state 1.

The complete Turing machine program for the bit inverter is

1. (1,0,1,1,R) Change the symbol 0 to 1.
2. (1,1,0,1,R) Change the symbol 1 to 0.

(We've added a comment to each instruction to explain its purpose.) Here's a sample computation using this machine, beginning with the string 1101 on the tape:

Using instruction 2,

Using instruction 2 again,

Using instruction 1,

Using instruction 2,

and the machine halts with the inverted string 0010 as output on the tape.

Practice Problems

Is the Turing machine shown here equivalent to the one shown in Figure 12.4, that is, will it produce the same output given the same input? Why or why not?

Answer

It is not equivalent. There is no transition from State 1 for an input symbol of 0, and no transition from State 2 for an input symbol of 1.

Explain exactly what would happen to the Turing machine of Practice Problem 1 if it were given a completely blank tape as input.

Answer

The machine would halt immediately because there are no instructions for what to do when looking at a blank cell.

Bit inversion might seem like a trivial task, but recall that in Chapter 4 we introduced an electronic device called a NOT gate that is essentially a bit inverter and is one of the components of a real computer.

Main content

Chapter Contents

12.5.2 A Parity Bit Machine

An extra bit, called an *odd parity bit*, can be attached to the end of a string of bits. The odd parity bit is set such that the total number of 1s in the whole string of bits, including the parity bit, is odd. Thus, if the string preceding the parity bit has an odd number of 1s, the parity bit is set to 0 so that there is still an odd number of 1s in the whole string. If the string preceding the parity bit has an even number of 1s, the parity bit is set to 1 so that the number of 1s in the whole string is odd. As an example, the following string of bits includes as its rightmost bit an odd parity bit:

1 1 0 0 0 1 0 1 0 1

The parity bit is set to 1 because there are four 1s (an even number) in the string before the parity bit; the total number of 1s is five (an odd number). Another example of odd parity is the string

1 0 1 1 0 0

where the parity bit (the rightmost bit) is a 0 because three 1s (an odd number) appear in the preceding string. Our job here is to write a Turing machine that, given a string of bits on its input tape, attaches the correct odd parity bit to the right end.

We know from Chapter 4 that information in electronic form is represented as strings of bits. **Parity bits** are used to detect errors that occur as a result of electronic interference when transmitting such strings (see Exercise 25, Chapter 4). If a single bit (or any odd number of bits) is changed from a 1 to 0 or from a 0 to 1, then the parity bit is incorrect, and the error can be detected. A correct copy of the information can then be retransmitted. Again, we are devising a Turing machine for a significant real-world task.

Our Turing machine must somehow "remember" whether the number of 1s processed so far is even or odd. We can use two states of the machine to represent these two conditions. Because the Turing machine begins in state 1, having read zero 1s so far (zero is an even number), we can let state 1 represent the even parity state, where an even number of 1s has been read so far. We'll let state 2 represent the odd parity state, where an odd number of 1s has been read so far.

We can read the input string from left to right. Until we get to the end of the bit string, the symbol printed should always be the same as the symbol read because none of the bits in the input string should change. But every time a 1 bit is read, the parity should change, from even to odd or from odd to even. In other words, the state should change from 1 to 2 or from 2 to 1. Reading a 0 bit does not affect the parity and therefore should not change the state. Thus, if we are in state 1 reading a 1, we want to go to state 2; if we are in state 1 reading a 0, we want to stay in state 1. If we are in state 2 reading a 1, we want to go to state 1; if we are in state 2 reading a 0, we want to stay in state 2.

When we come to the end of the input string (when we encounter the first blank cell), we write the parity bit, which is 1 if the machine is in state 1 (the even parity state) or 0 if the machine is in state 2 (the odd parity state). Then we want to halt, which is accomplished by going into state 3, for which there are no instructions. The state diagram for our odd parity bit machine is given in Figure 12.5.

Figure 12.5 State diagram for the odd parity bit machine

The Turing machine program is as follows:

1. (1,1,1,2,*R*) Even parity state reading 1, change state.

2. (1,0,0,1,*R*) Even parity state reading 0, don't change state.

3. (2,1,1,1,*R*) Odd parity state reading 1, change state.

4. (2,0,0,2,*R*) Odd parity state reading 0, don't change state.

5. (1,*b*,1,3,*R*) End of string in even parity state, write 1 and go to state 3.

6. (2,*b*,0,3,*R*) End of string in odd parity state, write 0 and go to state 3.

Let's do an example. The initial string is 101, which contains an even number of 1s. Therefore, we want to add a parity bit of 1 and have the final output be the string 1011. Because this final string contains three 1 bits, it has the correct parity. Here's the initial configuration:

Using instruction 1,

Using instruction 4,

Using instruction 3,

and finally using instruction 5 to write the parity bit, we get

whereupon the machine halts.

Main content

Chapter Contents

12.5.3 Machines for Unary Incrementing

Turing machines can be written to accomplish arithmetic using the nonnegative numbers 0, 1, 2, and so on. However, working with these numbers poses a problem we did not face with the bit inverter or the parity bit machine. In those examples, we were manipulating only bits (i.e., 0s and 1s), already part of the Turing machine alphabet of symbols. We can't put numbers like 2, 6, or 754 in cells of the Turing machine tape because these symbols are not part of the tape alphabet. Therefore, our first task is to find a way to encode such numbers using 0s and 1s. We could use binary representation, as a real computer does. Instead, let us agree on a simpler unary representation of numbers (**unary representation** means that we will use only *one* symbol, namely 1). In unary representation, any unsigned whole number *n* is encoded by a sequence of *n* 1s. Thus,

Number Turing	***Machine Representation***
0	1
1	11
2	111
3	1111

. .

. .

. .

(You may wonder why we don't simply use 1 to represent 1, 11 to represent 2, and so on. This scheme would mean using no 1s to represent 0, and then the machine could not distinguish a single 0 on the tape from nothing—all blanks—on the tape.)

Practice Problems

What should the output be when the parity bit Turing machine is run on the following input?

. . . *b* 1 1 0 1 *b* . . .

Answer

b 1 1 0 1 0 *b*

Now run the parity bit Turing machine on this tape and see whether you get the answer you expected from Practice Problem 1a.

Using this unary representation of numbers, let's write Turing machines to accomplish some basic arithmetic operations. We can write a Turing machine to add 1 to any number; such a machine is often called an *incrementor*. (Recall from Chapter 5 that the program counter in a Von Neumann architecture uses an incrementor to bump up the address for the next instruction to be executed.) Using the unary representation of numbers just described, we need only stay in state 1 and travel over the string of 1s to the right-hand end. When we encounter the first blank cell, we write a 1 in it and go to state 2, which has no instructions, in order to halt. Figure 12.6 shows the state diagram.

Figure 12.6 State diagram for the incrementer

The Turing machine for the incrementer is

1. (1,1,1,1,*R*)	Pass to the right over the 1s.
2. (1,b,1,2,*R*)	Add a single 1 at the right-hand end of the string and change to state 2.

Here's a quick sample computation:

at which point the machine halts. The output on the tape is the unary representation of the number 3. The machine has thus incremented the input, 2, to the output, 3.

Here is another algorithm to accomplish the same task. The preceding algorithm moved to the right-hand end of the string and added a 1. But the increment problem can also be solved by moving to the left-hand end of the string and adding a 1. The Turing machine program for this algorithm is

(1,1,1,1,*L*) Pass to the left over 1s.

(1,*b*,1,2,*L*) Add a single 1 at the left-hand end of the string and change to state 2.

If we apply this algorithm to the same input tape, the computation is

Once again, 2 has been incremented to 3. But whereas the first computation took four operations—that is, four applications of Turing machine instructions—the second computation took only two.

Let's compare these two algorithms in terms of their time and space efficiency. We'll take the execution of a single Turing machine instruction as a unit of work, so we measure the time used by a Turing machine algorithm by the number of instructions executed. The "space" a Turing machine algorithm takes on any given input is the number of nonblank cells on the tape that are used during the course of running the program. The input itself occupies some nonblank cells, so the interesting question is how many additional cells the algorithm uses in the course of its execution.

Suppose that the number 5 is to be incremented using algorithm 1. The initial input tape contains six 1s (the unary representation for 5). The machine moves to the right, over all the 1s on the tape, until it encounters the first blank cell. It writes a 1 into the blank cell and then halts. One instruction is executed for each move to the right. By the time the blank cell is reached, the first instruction has been executed six times. One final execution, this time of the second instruction, completes the task. Altogether, seven steps are required, two more than the number 5 we are incrementing. One "extra" step comes because of the unary representation, with its additional 1, and a second "extra" step writes over the blank cell. Therefore, it is easy to see that if the problem is to increment the number *n*, then steps would be required using algorithm 1. Algorithm 2 does a constant number of steps (two) no matter what the size of *n*. Both algorithms use cells on the tape: for the initial input and one more for incrementing. The algorithms are equivalent in space efficiency, but algorithm 2 is more time efficient. With an input such as 5, our example here, the difference in time efficiency between the two algorithms does not seem great. Figure 12.7 shows the steps required by algorithms 1 and 2 for larger problems. As the input gets larger, the difference in efficiency becomes more obvious. If our hypothetical Turing machine actually existed and could do, say, one step per second, then to increment the number 10,000, algorithm 1 would take 2 hours, 46 minutes, and 42 seconds. Algorithm 2 could do the same job in 2 seconds! This significant difference gives a definite edge to algorithm 2 as the preferable solution method for this problem. Using the notation of Chapter 3, algorithm 1 is a linear time algorithm, whereas algorithm 2 is a constant time algorithm.

Figure 12.7

Time efficiency for two Turing machine algorithms for incrementing

The Number to Be Incremented, *n*	Number of Steps Required	
	Algorithm 1	Algorithm 2
10	12	2

The Number to Be Incremented, *n*	Number of Steps Required	
	Algorithm 1	Algorithm 2
100	102	2
1,000	1,002	2
10,000	10,002	2

Although we can compare two Turing machine algorithms for the same task, we can't really compare the efficiency of a Turing machine algorithm with an algorithm that runs on a "real" computer. For one thing, the data representation is probably different (numbers aren't written in unary form). But more to the point, the basic unit of work is different. It takes many Turing machine operations to do a trivial task because the entire concept of a Turing machine is so simplistic. Turing machines, as we saw in our few examples, work by carefully moving, changing, and keeping track of individual 0s and 1s. Given such a limited range of activities, a Turing machine must exert a lot of effort to accomplish even mildly interesting tasks.

Main content

Chapter Contents

12.5.4 A Unary Addition Machine

A Turing machine can be written to add two numbers. Again using unary representation, let's agree to start with the two numbers on the tape separated by a single blank cell. When the Turing machine halts, the tape should contain the unary representation of the sum of the two numbers. The separating blank should be gone. If we erase the leftmost 1 and then fill in the separating blank with a 1, this has the effect of sliding the entire first number one cell to the right on the tape. Also, both numbers are originally written on the tape using unary representation, which means that there are two extra 1s on the tape, one for each number. When we are finished, we want to have only one extra 1, for the unary representation of the sum. Therefore, a second 1 should be removed from the tape. Our plan is to erase the two leftmost 1s on the tape, proceed rightward to the separating blank, and replace the blank with a 1.

For example, suppose we want to add . The original tape representation (rather than drawing the individual cells, we'll just show the tape contents) is

and the final representation should be the unary representation for the number 5,

Our algorithm will accomplish this transformation in stages. First, we erase the leftmost 1:

We then erase a second 1 from the left end (see Exercise 26 at the end of this chapter for the case when there is no "second 1"):

and then move to the right and change the separating blank to a 1:

The Turing machine begins in state 1, so we use that state to erase the leftmost 1 and move right, changing to state 2. The job of state 2 is to erase the second 1 and move right, changing to state 3. State 3 must move across any remaining 1s until it encounters the separating blank, which it changes to a 1 and then goes into a "halting state" with no instructions, state 4. A state diagram (Figure 12.8) illustrates the desired transitions to next states.

Figure 12.8State diagram for the addition machine

Here is the Turing machine program:

1. (1,1,*b*,2,*R*)	Erase the leftmost 1 and move right.
2. (2,1,*b*,3,*R*)	Erase the second 1 and move right.
3. (3,1,1,3,*R*)	Pass over any 1s until a blank is found.
4. (3,*b*,1,4,*R*)	Write a 1 over the blank and halt.

Try "running" this machine on the preceding addition problem to see exactly how it works.

Practice Problems

Set up the input and run the addition Turing machine to compute .
Answer

b 1 1 1 1 *b* 1 1 1 1 1 *b* becomes

b b b 1 1 1 1 1 1 1 1 1 *b*

Write a Turing machine that, when run on the tape

. . . b 1 1 1 0 b . . .

produces an output tape of

. . . b 1 1 1 0 1 b . . .

Answer

(1,1,1,1,*R*)
(1,0,0,1,*R*)
(1,*b*,1,2,*R*)

Does your machine from Problem 2a, when run on a tape containing k 1s followed by a single 0, produce k 1s followed by a single 0 and a single 1? If not, rewrite your machine so that it solves this more general problem.

Answer

The machine for part (a) also solves the general problem.

Main content

Chapter Contents

12.6 The Church–Turing Thesis

Just how good is the Turing machine as a model of the concept of an algorithm? We've already seen that any Turing machine exhibits the properties of an algorithm, and we've even produced Turing machine algorithms for a couple of important tasks. But perhaps we were judicious in our choice of tasks and happened to use those for which Turing machine instructions could be devised. We should ask whether there are other tasks that are "doable" by an algorithm but not "doable" by a Turing machine.

Of course, the answer to this question is yes. A Turing machine cannot program a DVR or shampoo hair, for example—tasks for which algorithms were given in Chapter 1. But suppose we limit the task to one for which the input and output can be represented symbolically, that is, using letters and numbers. Symbolic representation is, after all, how we traditionally record information such as names, addresses, telephone numbers, pay rates, yearly profits, temperatures, altitudes, growth rates, and so on. Taking a symbolic representation of information and manipulating it to produce a symbolic representation of other information covers a wide range of tasks, including everything done by "traditional" computing. Now let's ask a modified version of our previous question: Are there symbol manipulation tasks that are "doable" by an algorithm but not "doable" by a Turing machine?

The answer to this question is generally considered to be "No", as stated by the **Church–Turing thesis**, named for Alan Turing and another famous mathematician, Alonzo Church.

The Church–Turing thesis makes quite an extraordinary claim. It says that any symbol manipulation task that has an algorithmic solution can also be carried out by a Turing machine executing some set of Turing machine instructions. Processing the annual Internal Revenue Service records, for example, or calculating positions for global navigation satellite systems such as GPS can be done (according to this claim) using Turing machines. The thought of writing a Turing machine program to process IRS records is mind-boggling, but our examples may have convinced you that it is possible. Although such a program can be written, one can hardly imagine how many centuries it would take to execute, even with a very rapid "system clock." But the Church–Turing thesis says nothing about how efficiently the task will be done, only that it can be done by some Turing machine.

There are really two parts to writing a Turing machine for a symbol manipulation task. One part involves encoding symbolic information as strings of 0s and 1s so that it can

appear on Turing machine tapes. This is not difficult, and we know that real computers store all information, including graphical information, in binary. The other part is the heart of the challenge: Given that we can get the input encoded onto a Turing machine tape, can we write the Turing machine instructions that produce the encoded form of the correct output? Figure 12.9 illustrates the problem. The bottom arrow is the algorithmic solution to the symbol manipulation task we want to emulate. To perform this emulation, we must first encode the symbolic input into a bit string on a Turing machine tape (upward-pointing left arrow), write the Turing machine that solves the problem (top arrow), and, finally, decode the resulting bit string into symbolic output (downward-pointing right arrow). The Church–Turing thesis asserts that this process can always be done.

Figure 12.9 Emulating an algorithm by a Turing machine

The Turing Award

The most prestigious technical award in the field of computer science, presented by the Association for Computing Machinery, is the annual **Turing Award** (officially the A. M. Turing Award), named in honor of Alan Turing. It is sometimes called the "Nobel Prize" of computing, and is given to an individual selected for "major contributions of lasting importance to computing." Some of the individuals we've mentioned in this book have been recipients of the Turing Award, which was first given in 1966:

1971: John McCarthy (Chapter 10)

1977: John Backus (Chapters 10 and 11)

1983: Dennis Ritchie, Ken Thompson (Chapter 10)

2002: Ron Rivest, Adi Shamir, and Len Adleman (Chapter 8)

2004: Vinton Cerf and Robert Kahn (Chapter 7)

2005: Peter Naur (Chapter 11)

Other recipients of the award have made contributions in areas we have discussed or will discuss in later chapters:

1972: Edsger Dijkstra for fundamental contributions to programming as a high, intellectual challenge (Chapters 9 and 10)

1975: Allen Newell as one of the founding fathers of artificial intelligence (AI), beginning his work in this area in 1954 (Chapter 15)

1981: Edgar F. Codd for fundamental contributions to database management systems (Chapter 14)

1982: Stephen A. Cook for exploring the class of problems that in Chapter 3 we called "suspected intractable"

1986: John Hopcroft and Robert Tarjan for their work on analysis of algorithms (Chapter 3)

1990: Fernando J. Corbato for pioneering work on general-purpose, time-shared mainframe operating systems (Chapter 6)

1999: Frederick P. Brooks, Jr., for landmark contributions to computer architecture, operating systems, and software engineering (Chapters 5, 6, 9)

2008: Barbara Liskov for contributions to practical and theoretical foundations of programming language and system design (Chapters 6 and 9)

2010: Leslie Valiant for transformative contributions to the theory of computation (Chapter 12)

2013 Leslie Lamport for fundamental contributions to the theory and practice of distributed and concurrent systems (Chapter 6)

2014 Michael Stonebraker for fundamental contributions to the concepts and practices underlying modern database systems (Chapter 14)

2015 Martin Hellman and Whitfield Diffie for inventing and promulgating both asymmetric public-key cryptography, including its application to digital signatures, and a practical cryptographic key-exchange method (Chapter 8)

2016 Sir Tim Berners-Lee for inventing the World Wide Web, the first web browser, and the fundamental protocols and algorithms allowing the web to scale (Chapter 7)

> What exactly is a *thesis*? According to the dictionary, it is "a statement advanced for consideration and maintained by argument." That sounds less than convincing—hasn't the Church–Turing thesis been proved? No, and that's why it is called a thesis, not a theorem. Theorems are ideas that can be proved in a formal, mathematical way, such as "the sum of the interior angles of a triangle equals ." The Church–Turing thesis can never be proved because—despite all our talk about algorithms and their properties—the definition of an algorithm is still descriptive, not mathematical. It would be like trying to "prove" that an ideal day at the beach is sunny and . We might all agree on this, but we'll never be able to "prove" it. Well, then, the Church–Turing thesis makes a remarkable claim and can never be proved! Sounds pretty suspicious—what are the arguments on its behalf? There are two.
> One argument is that early on, when the thesis was first put forward, whenever computer science researchers described algorithmic solutions for tasks, they also tried to find Turing machines for those tasks. They were always successful; no one was ever able to put forth an algorithm for a task for which a Turing machine was not eventually found. This does not mean that no such task exists, but it lends weight to a body of evidence in support of the thesis.
>
> A second argument on behalf of the thesis is the fact that a number of other mathematicians attempted to find models for computing agents and algorithms. All of these proved to be equivalent to Turing machines and Turing machine programs in that whatever could be done by these other computing agents running their algorithms could also be done by a Turing machine running a Turing machine program, and vice versa. This suggests that the Turing machine captures all of these other ideas about "algorithms."

The Church–Turing thesis is now widely accepted by computer scientists. They no longer feel it necessary to write a Turing machine when they talk about an algorithmic computation. After describing an algorithm to carry out some task, they simply say, "Now let T be the Turing machine that does this task." You may make your own decision about the Church–Turing thesis, but in this book we will go along with convention and accept it as true. We therefore accept the Turing machine as an ultimate model of a computing agent and a Turing machine program as an ultimate model of an algorithm. We are saying that Turing machines define the limits of **computability**—that which can be done by symbol manipulation algorithms. What can be done by an algorithm is doable by a Turing machine, and what is not doable by a Turing machine cannot be done by an algorithm. In particular, if we find a symbol manipulation task that no Turing machine can perform (in its elementary way of moving around over a tape of 0s and 1s), then there is no algorithm for this task, and no real computer, no matter how sophisticated, will ever be able to do it either. That's why the Turing machine is so important. You can now see where this is all leading in terms of our search for a problem that has no algorithmic solution. Suppose we can find a (symbol manipulation) problem for which we can prove that no Turing machine exists to solve it. Then, because of the Church–Turing thesis, no algorithm exists to solve it either. The problem is an **uncomputable or unsolvable** problem.

If we pose a problem and try to construct a Turing machine to solve it but are not successful, that alone does not prove that no Turing machine exists. What we must do is actually prove that no one can ever find such a Turing machine—that it is not possible for a Turing machine to exist that solves this problem. It may appear that the introduction of Turing machines hasn't helped at all and that we are confronted by the same dilemma we faced at the beginning of this chapter. But Alan Turing, in the late 1930s, found such a problem and proved its unsolvability.

Main content

Chapter Contents

12.7 Unsolvable Problems

The problem Turing found is an ingenious one that itself involves Turing machine computations. A Turing machine that is executing an algorithm (a collection of Turing machine instructions) to solve some task must halt when begun on a tape containing input appropriate to that task. On other kinds of input, the Turing machine may not halt. It is easy enough for us to decide whether any specific configuration of a given Turing machine is a halting configuration. If a Turing machine program consists of the following four instructions:

(1,0,1,2,R)

(1,1,0,2,R)

(2,0,0,2,R)

(2,b,b,2,L)

then the configuration

is a halting configuration because there is no instruction of the form (2,1,-,-,-). It is also easy to see that this configuration will arise if the Turing machine is begun on the tape

Similarly, we can see that if the Turing machine is begun on the tape

then it will never halt. Instead, after the first step (clock tick), the machine will cycle forever between the two configurations

In a more complicated case, however, if we know the Turing machine program and we know the initial contents of the tape, then it may not be so easy to decide whether the Turing machine will eventually halt when begun on that tape. Of course, we can always simply execute the Turing machine—that is, carry out the instructions. We don't have all day to wait for the answer, so we'll set a time-out for our Turing machine system clock. Let's say we are willing to wait for 1,000 clock ticks. If we come to a halting configuration within the first 1,000 steps, then we know the answer: This Turing machine, running on this input tape, halts. But suppose we do not come to a halting configuration within the first 1,000 clock ticks. Can we say that the machine will never halt? Should we wait another 1,000 clock ticks? 10,000 clock ticks? Just running the Turing machine doesn't necessarily enable us to decide about halting. Here is the problem we propose to investigate:

Decide, given any collection of Turing machine instructions together with any initial tape contents, whether that Turing machine will ever halt if started on that tape.

This is a clear and unambiguous problem known as the **halting problem**. Does it have a Turing machine solution? Can we find one single Turing machine that will solve every instance of this problem—that is, one that will give us the answer "Yes, halts" or "No, never halts" for every (Turing machine, initial tape) pair?

This is an uncomputable problem; we will show that no Turing machine exists to solve this problem. Remember that we said it was not sufficient to look for such a machine and fail; we actually have to prove that no such machine can exist. The way to do this is to assume that such a Turing machine does exist and then show that this assumption leads to an impossible situation, so such a machine could not exist after all. This approach is called a **proof by contradiction**.

Assume, then, that *P* is a Turing machine that solves the halting problem. On the initial tape for *P* we have to put a description—using the binary digits 0 and 1—of a collection *T* of Turing machine instructions, as well as the initial tape content t on which those instructions run. This is the encoding part of Figure 12.9. Translating Turing machine instructions into binary form is tedious but not difficult. For example, we can use unary notation for machine states and tape symbols, designate the direction in which the read unit moves by 1 for *R* (right) and 11 for *L* (left), and separate the parts of a Turing machine instruction by 0s. Let's use *T** to symbolize the binary form of the collection *T* of Turing machine instructions. *P* is then run on a tape containing both *T** and t, so the initial tape for *P* looks like the following, where *T** and t may occupy many cells of the tape:

Our assumption is that P will always give us an answer ("Yes, halts" or "No, never halts"). *P*'s yes/no answer would be its output—what is written on the tape when *P* halts; therefore, *P* itself must always halt. Again, because the output is written on *P*'s tape, it also has to be in binary form, so let's say that a single 1 and all the rest blanks represent "yes," and a single 0 and all the rest blanks represent "no." This is the decoding part of Figure 12.9. To summarize:

Figure 12.10 is a pictorial representation of the actions of *P* when started on a tape containing *T** and *t*.

Figure 12.10 Hypothetical Turing machine *P* running on *T and *t***

When *P* halts with a single 1 on its tape, it does so because there are no instructions allowing *P* to proceed in its current state when reading 1. For example, *P* might be in state 9, and there is no instruction of the form

(9,1,-,-,-)

for machine *P*. Let's imagine adding more instructions to *P* to create a new machine *Q* that behaves just like *P* except that when it reaches this same configuration, it moves forever to the right on the tape instead of halting. To do this, pick some state not in *P*, say 52, and add the following two new instructions to *P*:

(9,1,1,52,*R*)

(52,*b*,*b*,52,*R*)

Figure 12.11 represents *Q*'s behavior when started on a tape containing *T** and *t*.

Figure 12.11 Hypothetical Turing machine *Q* running on *T and *t***

Finally, we'll create a new machine *S*. This machine first makes a copy of what appears on its input tape. (This is a doable, if tedious, task. The machine must "pick up" a 0 or 1 by going to a particular state, move to another part of the tape, and write a 0 or 1, depending on the state. It travels back and repeats the process; however, each time it picks up a 0 or 1, it must mark the tape with some marker symbol, say *X* for 0 and *Y* for 1, so that it doesn't try to pick it up again. At the end of the copying, the markers must be changed back to 0s and 1s.) After S is finished with its copying job, it uses the same instructions as machine *Q*.

Now what happens when machine *S* is run on a tape that contains *S**, the binary representation of *S*'s own instructions? *S* first makes a copy of *S** and then turns the computation over to *Q*, which is now running on a tape containing *S** and *S**. Figure 12.12 shows the result; this figure follows from Figure 12.11 where *T** and *t* are both *S**.

Figure 12.12 Hypothetical Turing machine *S* running on *S**

Figure 12.12 represents the behavior of *S* running on input *S**. The final outcome is either (left output)

or (right output)

(Perhaps you'll need to read this several times while looking at Figure 12.12 to convince yourself of what we have said.) We have backed ourselves into a corner here, but that's good. This is exactly the impossible situation we were hoping to find.

We assumed that there was a Turing machine that could solve the halting problem, and this assumption led to an impossible situation. The assumption is therefore incorrect, and no Turing machine can exist to solve the halting problem. Therefore, no algorithm can exist to solve this problem. The halting problem is an example of an unsolvable or uncomputable problem.

The halting problem seems rather abstract; perhaps we don't care whether it is unsolvable. However, real computer programs written in real programming languages to run on real computers are also symbol manipulation algorithms and, by the Church–Turing thesis, can be simulated by Turing machines. This means that the unsolvability of the halting problem has practical consequences. For example, we know that some C++, Java, and Python programs can get stuck in infinite loops. It would be nice to have a program that you could run ahead of time on any C++, Java, or Python program, together with its input, that would tell you, "Uh-oh, if you run this program on this input, it will get into an infinite loop," or "No problem, if you run this program on this input, it will eventually stop." The unsolvability of the halting problem says that no such program is possible. Other unsolvable problems, related to the halting problem, have the following practical consequences:

- No program can be written to decide whether any given program always stops eventually, no matter what the input.
- No program can be written to decide whether any two programs are equivalent (will produce the same output for all inputs).
- No program can be written to decide whether any given program run on any given input will ever produce some specific output.

  This last case means it is impossible to write a general automatic program tester—one that for any program can check whether, given input *A*, it produces correct output *B*. That is why program testing plays such an important role in the software development life cycle described in Chapter 9.

It is important to note, however, that these problems are unsolvable because of their generality. We are asking for *one* program that will decide something about *any* given program. It may be very easy to write a program *A* that can make a decision only about a specific program *B* by utilizing specialized properties of *B*. (*Analogy*: If I ask you to be ready to write "I love you" in English, you can do it; if I ask you to be ready to write "I love you" in any language I might later specify, you can't do it.)

Couldn't Do, Can't Do, Never Will Be Able to ...

Unsolvable problems are not confined to problems about running programs (Java programs, C++ programs, Python programs, or Turing machines). In Chapter 11, we talked about grammars that can be described in Backus-Naur Form (BNF) and about how a compiler parses a programming language statement by applying the rules of its grammar. We noted that ambiguous grammars are not suitable for programming languages because they can allow multiple interpretations of a statement. It would be nice to have a test (an algorithm) to decide whether any BNF grammar is ambiguous. This is an unsolvable problem—no such algorithm can exist. Deciding whether any two such grammars produce the same language is also unsolvable.

One of the earliest "decision problems" was posed by the British mathematician David Hilbert in 1900. Consider quadratic equations of the form

where a, b, and c are integers. We can easily decide whether any one such equation has integer solutions by applying the quadratic formula to solve the equation. But consider more general polynomial equations in several unknowns, such as

where the unknowns are x, y, z, and w and the coefficients (a, b, c, d, and e) are integers. Is there an algorithm to decide whether any such equation has integer solutions? In 1970, this problem was shown to be unsolvable.

Practice Problems

Explain how a proof by contradiction is done.

Answer

To prove that something is not true, assume that it is true and arrive at a contradiction. The assumption must then be wrong.

Write in your own words a description of the halting problem.
Write a paragraph that describes the proof of the unsolvability of the halting problem.

Main content

Chapter Contents

12.8 Conclusion

We began this chapter by proposing that there exist problems for which no solution algorithm exists. To prove such a statement, we looked for appropriate models of "computing agent" and "algorithm" that would enable us to concentrate on the fundamental nature of computation. After developing a list of properties inherent in any computing agent, we defined the Turing machine, noted that it incorporates these properties, and accepted it as a model of a computing agent. A Turing machine program incorporates the properties of an algorithm described in Chapter 1, so we accepted it as a model of an algorithm. Are these good models? Do they capture everything that is fundamental about computing and algorithms? After looking at a few Turing machines devised to do some simple tasks, we stated our position with a resounding yes in the form of the Church–Turing thesis: Not only is a Turing machine program an example of an algorithm, but also every symbolic manipulation algorithm can be done by a Turing

machine (we believe). This leap of faith—putting total confidence in Turing machine programs as models of algorithms—allows us to define the boundaries of computability. If it can't be done by a Turing machine, then it is not computable. Thus, the real value of Turing machines as models of computability is in exposing problems that are uncomputable—problems for which no algorithmic solution exists no matter how intelligent we are or how long we keep looking. As a practical matter, recognizing uncomputable problems certainly saves time; we are less likely to devote our lives to searching for algorithms that can never be. As a philosophical matter, it is important to know that computability has its limits, beyond which lies the great abyss of the uncomputable!

Main content

Chapter Contents

12.9 Summary of Level 4

In Level 4, we examined the use of procedural programming languages as a means for expressing algorithms at a high level of abstraction. Other high-level languages exist, including special-purpose languages, and those that follow other philosophies, such as functional languages and logic-based languages. Because algorithms written in high-level languages ultimately run on low-level hardware, program translators must convert from one level of algorithmic expression to another. We've looked at the series of tasks that a language compiler must perform to carry out this conversion. This final chapter of Level 4 proved that there are limits to computability—that there exist problems that can never be solved algorithmically.

With all of the hardware and software mechanisms in place to implement algorithmic problem solutions, we are ready to proceed to the next level—the level of applications—to see some of the ways in which computers (and algorithms) are being put to use.

The first application we examine relates very closely to what we have discussed in this chapter—building models. In Chapter 13, we will discuss simulation models that help us solve important problems such as predicting the weather, creating new medicines, tracking our economy, and designing safe and efficient airplanes.