What Is Computable?

> Let me remind you of some of the problems of computing machines.
>
> —RICHARD FEYNMAN (1965 Nobel Prize in physics)

We've now seen quite a number of clever, powerful, and beautiful algorithms—algorithms that turn the bare metal of a computer into a genius at your fingertips. In fact, it would be natural to wonder, based on the rhapsodic rhetoric in the preceding chapters, if there is *anything* that computers cannot do for us. The answer is absolutely clear if we limit ourselves to what computers can do *today*: there are plenty of useful tasks (mostly involving some form of artificial intelligence) that computers can't, at present, perform well. Examples include high-quality translation between languages like English and Chinese; automatically controlling a vehicle to drive safely and quickly in a busy city environment; and (as a teacher, this is a big one for me) grading students' work.

Yet, as we have seen already, it is often surprising what a really clever algorithm can achieve. Perhaps tomorrow, someone will invent an algorithm that will drive a car perfectly or do an excellent job of grading my students' work. These do seem like hard problems—but are they impossibly hard? Indeed, is there any problem at all that is so difficult, no one could ever invent an algorithm to solve it? In this chapter, we will see that the answer is a resounding yes: there *are* problems that can never be solved by computers. This profound fact—that some things are "computable" and others are not—provides an interesting counterpoint to the many algorithmic triumphs we've seen in the preceding chapters. No matter how many clever algorithms are invented in the future, there will always be problems whose answers are "uncomputable."

The existence of uncomputable problems is striking enough on its own, but the story of their discovery is even more remarkable.

The existence of such problems was known before the first electronic computers were ever built! Two mathematicians, one American and one British, independently discovered uncomputable problems in the late 1930s—several years before the first real computers emerged during the Second World War. The American was Alonzo Church, whose groundbreaking work on the theory of computation remains fundamental to many aspects of computer science. The Briton was none other than Alan Turing, who is commonly regarded as the single most important figure in the founding of computer science. Turing's work spanned the entire spectrum of computational ideas, from intricate mathematical theory and profound philosophy to bold and practical engineering. In this chapter, we will follow in the footsteps of Church and Turing on a journey that will eventually demonstrate the impossibility of using a computer for one particular task. That journey begins in the next section, with a discussion of bugs and crashes.

BUGS, CRASHES, AND THE RELIABILITY OF SOFTWARE

The reliability of computer software has improved tremendously in recent years, but we all know that it's still not a good idea to assume software will work correctly. Very occasionally, even high-quality, well-written software can do something it was not intended to do. In the worst cases, the software will "crash," and you lose the data or document you were working on (or the video game you were playing—very frustrating, as I know from my own experience). But as anyone who encountered home computers in the 1980s and 90s can testify, computer programs used to crash an awful lot more frequently than they do in the 21st century. There are many reasons for this improvement, but among the chief causes are the great advances in automated software checking tools. In other words, once a team of computer programmers has written a large, complicated computer program, they can use an automatic tool to check their newly created software for problems that might cause it to crash. And these automated checking tools have been getting better and better at finding potential mistakes.

So a natural question to ask would be: will the automated software-checking tools ever get to the point where they can detect all potential problems in all computer programs? This would certainly be nice, since it would eliminate the possibility of software crashes once and for all. The remarkable thing that we'll learn in this chapter is that this software nirvana will never be attained: it is provably impossible

for any software-checking tool to detect all possible crashes in all programs.

It's worth commenting a little more on what it means for something to be "provably impossible." In most sciences, like physics and biology, scientists make hypotheses about the way certain systems behave, and conduct experiments to see if the hypotheses are correct. But because the experiments always have some amount of uncertainty in them, it's not possible to be 100% certain that the hypotheses were correct, even after a very successful experiment. However, in stark contrast to the physical sciences, it *is* possible to claim 100% certainty about some of the results in mathematics and computer science. As long as you accept the basic axioms of mathematics (such as $1 + 1 = 2$), the chain of deductive reasoning used by mathematicians results in absolute certainty that various other statements are true (for example, "any number that ends in a 5 is divisible by 5"). This kind of reasoning does not involve computers: using only a pencil and paper, a mathematician can prove indisputable facts.

So, in computer science, when we say that "X is provably impossible," we don't just mean that X appears to be very difficult, or might be impossible to achieve in practice. We mean that it is 100% certain that X can never be achieved, because someone has proved it using a chain of deductive, mathematical reasoning. A simple example would be "it is provably impossible that a multiple of 10 ends with the digit 3." Another example is the final conclusion of this chapter: it is provably impossible for an automated software-checker to detect all possible crashes in all computer programs.

PROVING THAT SOMETHING ISN'T TRUE

Our proof that crash-detecting programs are impossible is going to use a technique that mathematicians call *proof by contradiction.* Although mathematicians like to lay claim to this technique, it's actually something that people use all the time in everyday life, often without even thinking about it. Let me give you a simple example.

To start with, we need to agree on the following two facts, which would not be disputed by even the most revisionist of historians:

1. The U.S. Civil War took place in the 1860s.
2. Abraham Lincoln was president during the Civil War.

Now, suppose I made the statement: "Abraham Lincoln was born in 1520." Is this statement true or false? Even if you knew nothing whatsoever about Abraham Lincoln, apart from the two facts above, how could you quickly determine that my statement is false?

Most likely, your brain would go through a chain of reasoning similar to the following: (i) No one lives for more than 150 years, so if Lincoln was born in 1520, he must have died by 1670 at the absolute latest; (ii) Lincoln was president during the Civil War, so the Civil War must have occurred before he died—that is, before 1670; (iii) but that's impossible, because everyone agrees the Civil War took place in the 1860s; (iv) *therefore,* Lincoln could not possibly have been born in 1520.

But let's try to examine this reasoning more carefully. Why is it valid to conclude that the initial statement was false? It is because we proved that this claim contradicts some other fact that is known to be true. Specifically, we proved that the initial statement implies the Civil War occurred before 1670—which contradicts the known fact that the Civil War took place in the 1860s.

Proof by contradiction is an extremely important technique, so let's do a slightly more mathematical example. Suppose I made the following claim: "On average, a human heart beats about 6000 times in 10 minutes." Is this claim true or false? You might immediately be suspicious, but how would you go about proving to yourself that it is false? Spend a few seconds now trying to analyze your thought process before reading on.

Again, we can use proof by contradiction. First, assume for argument's sake that the claim is true: human hearts average 6000 beats in 10 minutes. If that were true, how many beats would occur in just one minute? On average, it would be 6000 divided by 10, or 600 beats per minute. Now, you don't have to be a medical expert to know that this is far higher than any normal pulse rate, which is somewhere between 50 and 150 beats per minute. So the original claim contradicts a known fact and must be false: it is *not* true that human hearts average 6000 beats in 10 minutes.

In more abstract terminology, proof by contradiction can be summarized as follows. Suppose you suspect that some statement S is false, but you would like to prove beyond doubt that it is false. First, you assume that S is true. By applying some reasoning, you work out that some other statement, say T, must also be true. If, however, T is known to be false, you have arrived at a contradiction. This proves that your original assumption (S) must have been false.

A mathematician would state this much more briefly, by saying something like "S implies T, but T is false, therefore S is false." That is proof by contradiction in a nutshell. The following table shows how to connect this abstract version of proof by contradiction with the two examples above:

	First example	Second example
S (original statement)	Lincoln was born in 1520	Human heart beats 6000 times in 10 minutes
T (implied by *S*, but known to be false)	Civil War occurred before 1670	Human heart beats 600 times in 1 minute
Conclusion: *S* is false	Lincoln was *not* born in 1520	Human heart does *not* beat 6000 times in 10 minutes

For now, our detour into proof by contradiction is finished. The final goal of this chapter will be to prove, by contradiction, that a program which detects all possible crashes in other programs cannot exist. But before marching on toward this final goal, we need to gain familiarity with some interesting concepts about computer programs.

PROGRAMS THAT ANALYZE OTHER PROGRAMS

Computers slavishly follow the exact instructions in their computer programs. They do this completely deterministically, so the output of a computer program is exactly the same every time you run it. Right? Or wrong? In fact, I haven't given you enough information to answer this question. It's true that certain simple computer programs produce exactly the same output every time they are run, but most of the programs we use every day look very different every time we run them. Consider your favorite word processing program: does the screen look the same every time it starts up? Of course not—it depends on what document you opened. If I use Microsoft Word to open the file "address-list.docx," the screen will display a list of addresses that I keep on my computer. If I use Microsoft Word to open the file "bank-letter.docx," I see the text of a letter I wrote to my bank yesterday. (If the ".docx" here seems mysterious to you, check out the box on the facing page to find out about file name extensions.)

Let's be very clear about one thing: in both cases, I'm running exactly the same computer program, which is Microsoft Word. It's just that the *inputs* are different in each case. Don't be fooled by the fact that all modern operating systems let you run a computer

Throughout this chapter, I'll be using file names like "abcd.txt." The part after the period is called the "extension" of the file name—in this case, the extension of "abcd.txt" is "txt." Most operating systems use the extension of a file name to decide what type of data the file contains. For example, a ".txt" file typically contains plain text, a ".html" file typically contains a web page, and a ".docx" file contains a Microsoft Word document. Some operating systems hide these extensions by default, so you might not see them unless you turn off the "hide extensions" feature in your operating system. A quick web search for "unhide file extensions" will turn up instructions on how to do this.

Some technical details about file name extensions.

program by double-clicking on a document. That is just a convenience that your friendly computer company (most likely Apple or Microsoft) has provided you. When you double-click on a document, a certain computer program gets run, and that program uses the document as its input. The output of the program is what you see on the screen, and naturally it depends on what document you clicked on.

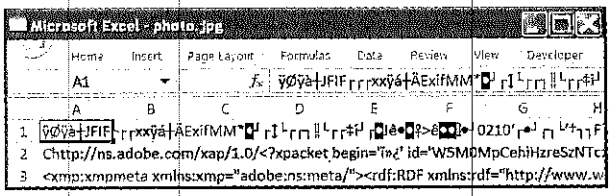
In reality, the input and output of computer programs is quite a bit more complex than this. For instance, when you click on menus or type into a program, you are giving it additional input. And when you save a document or any other file, the program is creating additional output. But to keep things simple, let's imagine that programs accept exactly one input, which is a file stored on your computer. And we'll also imagine that programs produce exactly one output, which is a graphical window on your monitor.

Unfortunately, the modern convenience of double-clicking on files clouds an important issue here. Your operating system uses various clever tricks to guess which program you would like to run whenever you double-click on a file. But it's very important to realize that it's possible to open *any* file using *any* program. Or to put it another way, you can run any program using any file as its input. How can you do this? The box on the next page lists several methods you can try. These methods will not work on all operating systems, or on all choices of input file—different operating systems launch programs in different ways, and they sometimes limit the choice of input file due to security concerns. Nevertheless, I strongly urge you to experiment for a few minutes with your own computer, to convince yourself that you can run your favorite word processing program with various different types of input files.

Here are three ways you could run the program Microsoft Word using `stuff.txt` as the input file:

- Right-click on `stuff.txt`, choose "Open with...," and select Microsoft Word.
- First, use the features of your operating system to place a shortcut to Microsoft Word on your desktop. Then drag `stuff.txt` onto this Microsoft Word shortcut.
- Open the Microsoft Word application directly, go to the "File" menu, choose the "Open" command, make sure the option to display "all files" is selected, then choose `stuff.txt`.

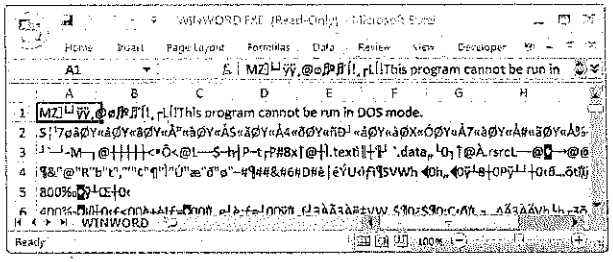
Various ways of running a program with a particular file as its input.



Microsoft Excel run with "photo.jpg" as its input. The output is garbage, but the important point is that you can, in principle, run any program on any input you want.

Obviously, you can get rather unexpected results if you open a file using a program it was not intended for. In the figure above, you can see what happens if I open the picture file "photo.jpg" with my spreadsheet program, Microsoft Excel. In this case, the output is garbage and is no use to anyone. But the spreadsheet program did run, and did produce some output.

This may already seem ridiculous, but we can take the craziness one step further. Remember that computer programs are themselves stored on the computer's disk as files. Often, these programs have a name that ends in ".exe," which is short for "executable"—this just means that you can "execute," or run, the program. So because computer programs are just files on the disk, we can feed one computer program as input to another computer program. As one specific example, the Microsoft Word program is stored on my computer as the file "WINWORD.EXE." So by running my spreadsheet program with the file WINWORD.EXE as input, I can produce the wonderful garbage you see in the figure on the facing page.

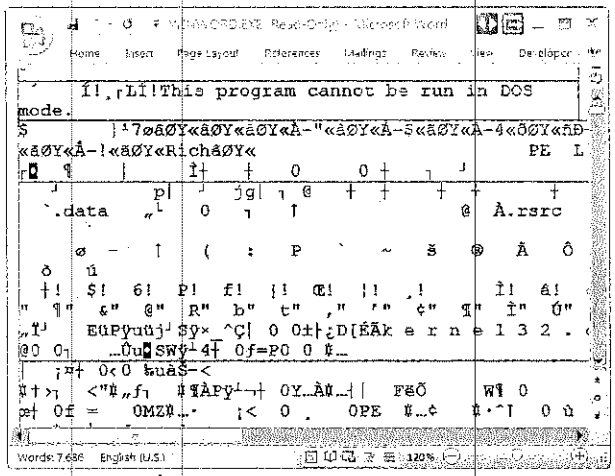


Microsoft Excel examines Microsoft Word. When Excel opens the file WINWORD.EXE, the result is—unsurprisingly—garbage.

Again, it would be well worth trying this experiment for yourself. To do that, you will need to locate the file WINWORD.EXE. On my computer, WINWORD.EXE lives in the folder "C:\Program Files\Microsoft Office\Office12," but the exact location depends on what operating system you are running and what version of Microsoft Office is installed. You may also need to enable the viewing of "hidden files" before you can see this folder. And, by the way, you can do this experiment (and one below) with any spreadsheet and word processing programs, so you don't need Microsoft Office to try it.

One final level of stupidity is possible here. What if we ran a computer program *on itself*? For example, what if I ran Microsoft Word, using the file WINWORD.EXE as input? Well, it's easy enough to try this experiment. The figure on the next page shows the result when I try it on my computer. As with the previous few examples, the program runs just fine, but the output on the screen is mostly garbage. (Once again, try it for yourself.)

So, what is the point of all this? The purpose of this section was to acquaint you with some of the more obscure things you can do when running a program. By now, you should be comfortable with three slightly strange ideas that will be very important later. First, there is the notion that any program can be run with any file as input, but the resulting output will usually be garbage unless the input file was intentionally produced to work with the program you chose to run. Second, we found out that computer programs are stored as files on computer disks, and therefore one program can be run with another program as its input file. Third, we realized that a computer program can be run using its own file as the input. So far, the second and third activities always produced garbage, but in the next section we will see a fascinating instance in which these tricks finally bear some fruit.



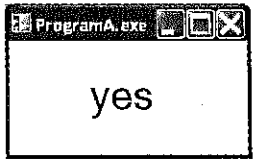
Microsoft Word examines itself. The open document is the file WINWORD.EXE, which is the actual computer program run when you click on Microsoft Word.

SOME PROGRAMS CAN'T EXIST

Computers are great at executing simple instructions—in fact, modern computers execute simple instructions billions of times every second. So you might think that any task that could be described in simple, precise English could be written down as a computer program and executed by a computer. My objective in this section is to convince you that the opposite is true: there are some simple, precise English statements that are literally impossible to write down as a computer program.

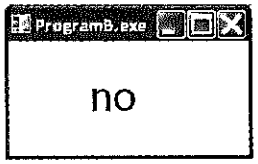
Some Simple Yes–No Programs

To keep things as simple as possible in this section, we will consider only a very boring set of computer programs. We'll call these "yes–no" programs, because the only thing these programs can do is pop up a single dialog box, and the dialog box can contain either the word "yes" or the word "no." For example, a few minutes ago I wrote a computer program called ProgramA.exe, which does nothing but produce the following dialog box:



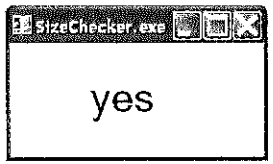
Note that by looking in the title bar of the dialog box, you can see the name of the program that produced this output—in this case, ProgramA.exe.

I also wrote a different computer program called ProgramB.exe, which outputs "no" instead of "yes":

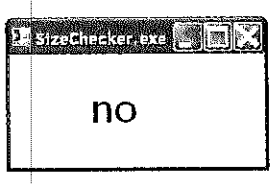


ProgramA and ProgramB are extremely simple—so simple, in fact, that they do not require any input (if they do receive input, they ignore it). In other words, they are examples of programs that really do behave exactly the same every time they are run, regardless of any input they may be given.

As a more interesting example of one of these yes–no programs, I created a program called SizeChecker.exe. This program takes one file as input and outputs "yes" if that file is bigger than 10 kilobytes and otherwise outputs "no." If I right-click on a 50-megabyte video file (say, mymovie.mpg), choose "Open with...," and select SizeChecker.exe, I will see the following output:



On the other hand, if I run the same program on a small 3-kilobyte e-mail message (say, myemail.msg), I will, of course, see a different output:



Therefore, SizeChecker.exe is an example of a yes–no program that sometimes outputs "yes" and sometimes "no."

Now consider the following slightly different program, which we'll call NameSize.exe. This program examines the *name* of its input file. If the file name is at least one character long, NameSize.exe outputs "yes"; otherwise, it outputs "no." What are the possible outputs of this program? Well, by definition, the name of any input file is at least one character long (otherwise, the file would have no name at all, and you couldn't select it in the first place). Therefore, NameSize.exe will always output "yes," regardless of its input.

By the way, the last few programs mentioned above are our first examples of programs that do not produce garbage when they are given other programs as input. For example, it turns out that the size of the file NameSize.exe is only about 8 kilobytes. So if I run SizeChecker.exe with NameSize.exe as the input, the output is "no" (because NameSize.exe is not more than 10 kilobytes). We can even run SizeChecker.exe on itself. The output this time is "yes," because it turns out that SizeChecker.exe is larger than 10 kilobytes—about 12 kilobytes, in fact. Similarly, we could run NameSize.exe with itself as input; the output would be "yes" since the file name "NameSize.exe" contains at least one character. All of the yes–no programs we have discussed this far are admittedly rather dull, but it's important to understand their behavior, so work through the table on the facing page line by line, making sure you agree with each output.

AlwaysYes.exe: A Yes–No Program That Analyzes Other Programs

We're now in a position to think about some much more interesting yes–no programs. The first one we'll investigate is called "AlwaysYes.exe." This program examines the input file it is given and outputs "yes" if the input file is itself a yes–no program that *always* outputs "yes." Otherwise, the output of AlwaysYes.exe is "no." Note that AlwaysYes.exe works perfectly well on any kind of input file. If you give it an input that isn't an executable program (e.g., address-list.docx), it will output "no." If you give it an input that *is* an executable program, but isn't a yes–no program (e.g., WINWORD.EXE),

program run	input file	output
ProgramA.exe	address-list.docx	yes
ProgramA.exe	ProgramA.exe	yes
ProgramB.exe	address-list.docx	no
ProgramB.exe	ProgramA.exe	no
SizeChecker.exe	mymovie.mpg (50MB)	yes
SizeChecker.exe	myemail.msg (3KB)	no
SizeChecker.exe	NameSize.exe (8KB)	no
SizeChecker.exe	SizeChecker.exe (12KB)	yes
NameSize.exe	mymovie.mpg	yes
NameSize.exe	ProgramA.exe	yes
NameSize.exe	NameSize.exe	yes

The outputs of some simple yes–no programs. Note the distinction between programs that *always* output "yes," regardless of their input (e.g., ProgramA.exe, NameSize.exe), and programs that output "no" either sometimes (e.g., SizeChecker.exe) or always (e.g., ProgramB.exe).

it will output "no." If you give it an input that is a yes–no program, but it's a program that sometimes outputs "no," then AlwaysYes.exe outputs "no." The only way that AlwaysYes.exe can output "yes" is if you input a yes–no program that *always* outputs "yes," regardless of its input. In our discussions so far, we've seen two examples of programs like this: ProgramA.exe, and NameSize.exe. The table on the next page summarizes the output of AlwaysYes.exe on various different input files, including the possibility of running AlwaysYes.exe on itself. As you can see in the last line of the table, AlwaysYes.exe outputs "no" when run on itself, because there are at least some input files on which it outputs "no."

In the next-to-last line of this table, you may have noticed the appearance of a program called Freeze.exe, which has not been described yet. Freeze.exe is a program that does one of the most annoying things a computer program can do: it "freezes" (no matter what its input is). You have probably experienced this yourself, when a video game or an application program seems to just lock up (or "freeze") and refuses to respond to any more input whatsoever. After that, your only option is to kill the program. If that doesn't work, you might even need to turn off the power (sometimes, when using a

AlwaysYes.exe outputs

input file	output
address-list.docx	no
mymovie.mpg	no
WINWORD.EXE	no
ProgramA.exe	yes
ProgramB.exe	no
NameSize.exe	yes
SizeChecker.exe	no
Freeze.exe	no
AlwaysYes.exe	no

The outputs of AlwaysYes.exe for various inputs. The only inputs that produce a "yes" are yes-no programs that *always* output "yes"—in this case, ProgramA.exe and NameSize.exe.

laptop, this requires removing the batteries!) and reboot. Computer programs can freeze for a variety of different reasons. Sometimes, it is due to "deadlock," which was discussed in chapter 8. In other cases, the program might be busy performing a calculation that will never end—for example, repeatedly searching for a piece of data that is not actually present.

In any case, we don't need to understand the details about programs that freeze. We just need to know what AlwaysYes.exe should do when it's given such a program as input. In fact, AlwaysYes.exe was defined carefully so that the answer is clear: AlwaysYes.exe outputs "yes" if its input always outputs "yes"; otherwise, it outputs "no." Therefore, when the input is a program like Freeze.exe, AlwaysYes.exe must output "no," and this is what we see in the next-to-last line of the table above.

YesOnSelf.exe: A Simpler Variant of AlwaysYes.exe

It may have already occurred to you that AlwaysYes.exe is a rather clever and useful program, since it can analyze other programs and predict their outputs. I will admit that I didn't actually write this program—I just described how it would behave, if I had written it. And now I am going to describe another program, called YesOnSelf.exe. This program is similar to AlwaysYes.exe, but simpler.

YesOnSelf.exe outputs

input file	output
address-list.docx	no
mymovie.mpg	no
WINWORD.EXE	no
ProgramA.exe	yes
ProgramB.exe	no
NameSize.exe	yes
SizeChecker.exe	yes
Freeze.exe	no
AlwaysYes.exe	no
YesOnSelf.exe	???

The outputs of YesOnSelf.exe for various inputs. The only inputs that produce a "yes" are yes-no programs that output "yes" when given themselves as input—in this case, ProgramA.exe, NameSize.exe, and SizeChecker.exe. The last line in the table is something of a mystery, since it seems as though either possible output might be correct. The text discusses this in more detail.

Instead of outputting "yes" if the input file *always* outputs "yes," YesOnSelf.exe outputs "yes" if the input file outputs "yes" *when run on itself*; otherwise, YesOnSelf.exe outputs "no." In other words, if I provide SizeChecker.exe as the input to YesOnSelf.exe, then YesOnSelf.exe will do some kind of analysis on SizeChecker.exe to determine what the output is when SizeChecker.exe is run with SizeChecker.exe as the input. As we already discovered (see the table on page 185), the output of SizeChecker.exe on itself is "yes." Therefore, the output of YesOnSelf.exe on SizeChecker.exe is "yes" too. You can use the same kind of reasoning to fill in the outputs of YesOnSelf.exe for various other inputs. Note that if the input file isn't a yes-no program, then YesOnSelf.exe automatically outputs "no." The table above shows some of the outputs for YesOnSelf.exe—try to verify that you understand each line of this table, since it's very important to understand the behavior of YesOnSelf.exe before reading on.

We need to note two more things about this rather interesting program, YesOnSelf.exe. First, take a look at the last line in the table above. What should be the output of YesOnSelf.exe, when it is given

the file YesOnSelf.exe as an input? Luckily, there are only two possibilities, so we can consider each one in turn. If the output is "yes," we know that (according to the definition of YesOnSelf.exe), YesOnSelf.exe should output "yes" when run on itself. This is a bit of a tongue twister, but if you reason through it carefully, you'll see that everything is perfectly consistent, so you might be tempted to conclude that "yes" is the right answer.

But let's not be too hasty. What if the output of YesOnSelf.exe when run on itself happened to be "no"? Well, it would mean that (again, according to the definition of YesOnSelf.exe) YesOnSelf.exe should output "no" when run on itself. Again, this statement is perfectly consistent! It seems like YesOnSelf.exe can actually choose what its output should be. As long as it sticks to its choice, its answer will be correct. This mysterious freedom in the behavior of YesOnSelf.exe will soon turn out to be the innocuous tip of a rather treacherous iceberg, but for now we will not explore this issue further.

The second thing to note about YesOnSelf.exe is that, as with the slightly more complicated AlwaysYes.exe, I didn't actually write the program. All I did was describe its behavior. However, note that if we assume I *did* write AlwaysYes.exe, then it would be easy to create YesOnSelf.exe. Why? Because YesOnSelf.exe is simpler than AlwaysYes.exe: it only has to examine one possible input, rather than all possible inputs.

AntiYesOnSelf.exe: The Opposite of YesOnSelf.exe

It's time to take a breath and remember where we are trying to get to. The objective of this chapter is to prove that a crash-finding program cannot exist. But our immediate objective is less lofty. In this section, we are merely trying to find an example of some program that cannot exist. This will be a useful steppingstone on the way to our ultimate goal, because once we've seen *how* to prove that a certain program can't exist, it will be reasonably straightforward to use the same technique on a crash-finding program. The good news is, we are very close to this steppingstone goal. We will investigate one more yes-no program, and the job will be done.

The new program is called "AntiYesOnSelf.exe." As its name suggests, it is very similar to YesOnSelf.exe—in fact, it is identical, except that its outputs are reversed. So if YesOnSelf.exe would output "yes" given a certain input, then AntiYesOnSelf.exe would output "no" on that same input. And if YesOnSelf.exe outputs "no" on an input, AntiYesOnSelf.exe outputs "yes" on that input.

> Whenever the input file is a yes-no program, AntiYesOnSelf.exe answers the question:
>
> **Will the input program, when run on itself, output "no"?**

A concise description of the behavior of AntiYesOnSelf.exe.

Although that amounts to a complete and precise definition of AntiYesOnSelf.exe's behavior, it will help to spell out the behavior even more explicitly. Recall that YesOnSelf.exe outputs "yes" if its input would output "yes" when run on itself, and "no" otherwise. Therefore, AntiYesOnSelf.exe outputs "no" if its input would output "yes" when run on itself, and "yes" otherwise. Or to put it another way, AntiYesOnSelf.exe answers the following question about its input: "Is it true that the input file, when run on itself, will not output 'yes'?"

Admittedly, this description of AntiYesOnSelf.exe is another tongue twister. You might think it would be simpler to rephrase it as "Will the input file, when run on itself, output 'no'?" Why would that be incorrect? Why do we need the legalese about not outputting "yes," instead of the simpler statement about outputting "no"? The answer is that programs can sometimes do something other than output "yes" or "no." So if someone tells us that a certain program does not output "yes," we can't automatically conclude that it outputs "no." For example, it might output garbage, or even freeze. However, there is one particular situation in which we can draw a stronger conclusion: if we are told in advance that a program is a yes-no program, then we know that the program never freezes and never produces garbage—it always terminates and produces the output "yes" or "no." Therefore, *for yes-no programs*, the legalese about not outputting "yes" is equivalent to the simpler statement about outputting "no."

Finally, therefore, we can give a very simple description of AntiYesOnSelf.exe's behavior. Whenever the input file is a yes-no program, AntiYesOnSelf.exe answers the question "Will the input program, when run on itself, output 'no'?" This formulation of AntiYesOnSelf.exe's behavior will be so important later that I have put it in a box above.

Given the work we've done already to analyze YesOnSelf.exe, it is particularly easy to draw up a table of outputs for AntiYesOnSelf.exe. In fact, we can just copy the table on page 187, switching all the

AntiYesOnSelf.exe outputs

input file	output
address-list.docx	yes
mymovie.mpg	yes
WINWORD.EXE	yes
ProgramA.exe	no
ProgramB.exe	yes
NameSize.exe	no
SizeChecker.exe	no
Freeze.exe	yes
AlwaysYes.exe	yes
AntiYesOnSelf.exe	???

The outputs of AntiYesOnSelf.exe for various inputs. By definition, AntiYesOnSelf.exe produces the opposite answer to YesOnSelf.exe, so this table—except for its last row—is identical to the one on page 187, but with the outputs switched from "yes" to "no" and vice versa. The last row presents a grave difficulty, as discussed in the text.

outputs from "yes" to "no" and vice versa. Doing this produces the table above. As usual, it would be a good idea to run through each line in this table, and verify that you agree with the entries in the output column. Whenever the input file is a yes-no program, you can use the simple formulation in the box on the previous page, instead of working through the more complicated one given earlier.

As you can see from the last row of the table, a problem arises when we try to compute the output of AntiYesOnSelf.exe on itself. To help us analyze this, let's further simplify the description of AntiYesOnSelf.exe given in the box on the previous page: instead of considering all possible yes-no programs as inputs, we'll concentrate on what happens when AntiYesOnSelf.exe is given itself as input. So the question in bold in that box, "Will the input program, ...," can be rephrased as "Will AntiYesOnSelf.exe, ..."—because the input program *is* AntiYesOnSelf.exe. This is the final formulation we will need, so it is also presented in a box on the facing page.

Now we're ready to work out the output of AntiYesOnSelf.exe on itself. There are only two possibilities ("yes" and "no"), so it shouldn't be too hard to work through this. We'll just deal with each of the cases in turn:

> AntiYesOnSelf.exe, when given itself as input, answers the question:
>
> **Will AntiYesOnSelf.exe, when run on itself, output "no"?**

A concise description of the behavior of AntiYesOnSelf.exe when given itself as input. Note that this box is just a simplified version of the box on page 189, specialized to the single case that the input file is AntiYesOnSelf.exe.

Case 1 (output is "yes"): If the output is "yes," then the answer to the question in bold in the box above is "no." But the answer to the bold question is, by definition, the output of AntiYesOnSelf.exe (read the whole box again to convince yourself of this)—and therefore, the output must be "no." To summarize, we just proved that if the output is "yes," then the output is "no." Impossible! In fact, we have arrived at a *contradiction.* (If you're not familiar with the technique of proof by contradiction, this would be a good time to go back and review the discussion of this topic earlier in this chapter. We'll be using the technique repeatedly in the next few pages.) Because we obtained a contradiction, our assumption that the output is "yes" must be invalid. We have proved that the output of AntiYesOnSelf.exe, when run on itself, cannot be "yes." So let's move on to the other possibility.

Case 2 (output is "no"): If the output is "no," then the answer to the question in bold in the box above is "yes." But, just as in case 1, the answer to the bold question is, by definition, the output of AntiYesOnSelf.exe—and, therefore, the output must be "yes." In other words, we just proved that if the output is "no," then the output is "yes." Once again, we have obtained a contradiction, so our assumption that the output is "no" must be invalid. We have proved that the output of AntiYesOnSelf.exe, when run on itself, cannot be "no."

So what now? We have eliminated the only two possibilities for the output of AntiYesOnSelf.exe when run on itself. This too is a contradiction: AntiYesOnSelf.exe was defined to be a yes-no program—a program that always terminates and produces one of the two outputs "yes" or "no." And yet we just demonstrated a particular input for which AntiYesOnSelf.exe does not produce either of these outputs! This contradiction implies that our initial assumption was false:

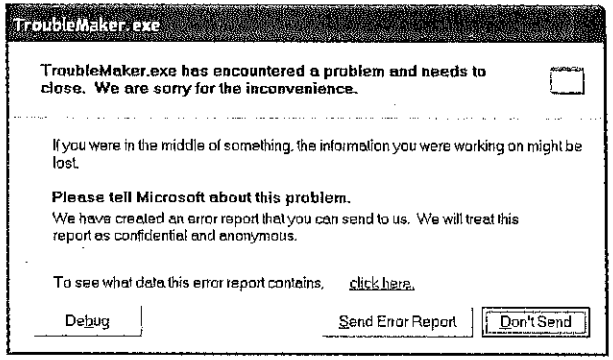
thus, it is *not* possible to write a yes-no program that behaves like AntiYesOnSelf.exe.

Now you will see why I was very careful to be honest and admit that I did not actually write any of the programs AlwaysYes.exe, YesOnSelf.exe, or AntiYesOnSelf.exe. All I did was describe how these programs would behave if I did write them. In the last paragraph, we used proof by contradiction to show that AntiYesOnSelf.exe cannot exist. But we can prove even more: the existence of AlwaysYes.exe and YesOnSelf.exe is also impossible! Why is this? As you can probably guess, proof by contradiction is again the key tool. Recall how we discussed, on page 188, that if AlwaysYes.exe existed, it would be easy to make a few small changes to it and produce YesOnSelf.exe. And if YesOnSelf.exe existed, it would be extremely easy to produce AntiYesOnSelf.exe, since we just have to reverse the outputs ("yes" instead of "no," and vice versa). In summary, if AlwaysYes.exe exists, then so does AntiYesOnSelf.exe. But we already know that AntiYesOnSelf.exe can't exist, and, therefore, AlwaysYes.exe can't exist either. The same argument shows that YesOnSelf.exe is also an impossibility.

Remember, this whole section was just a steppingstone toward our final goal of proving that crash-finding programs are impossible. The more modest goal in this section was to give some examples of programs that cannot exist. We've achieved this by examining three different programs, each of which is impossible. Of these three, the most interesting is AlwaysYes.exe. The other two are rather obscure, in that they concentrate on the behavior of programs that are given themselves as input. AlwaysYes.exe, on the other hand, is a very powerful program, since if it existed, it could analyze any other program and tell us whether that program always outputs "yes." But as we've now seen, no one will ever be able to write such a clever and useful-sounding program.

THE IMPOSSIBILITY OF FINDING CRASHES

We are finally ready to begin a proof about a program that successfully analyzes other programs and determines whether or not they crash: specifically, we will be proving that such a program cannot exist. After reading the last few pages, you have probably guessed that we will be using proof by contradiction. That is, we will start off by assuming that our holy grail exists: there is some program called CanCrash.exe which can analyze other programs and tell us whether or not they can crash. After doing some strange, mysterious, and delightful things to CanCrash.exe, we will arrive at a contradiction.



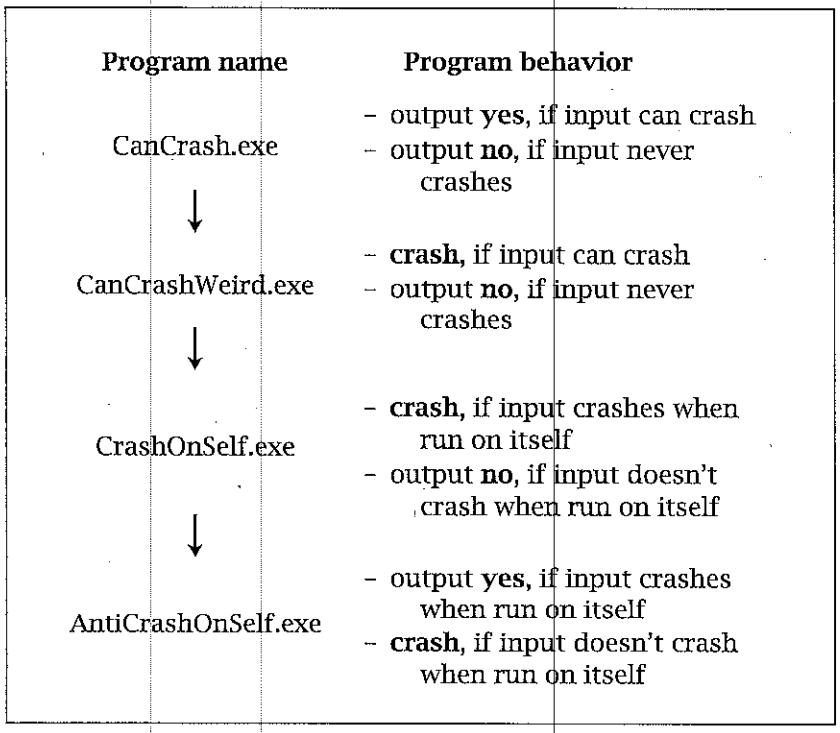
The result of a crash on one particular operating system. Different operating systems handle crashes in different ways, but we all know one when we see one. This TroubleMaker.exe program was deliberately written to cause a crash, demonstrating that intentional crashes are easy to achieve.

One of the steps in the proof requires us to take a perfectly good program and alter it so that it deliberately crashes under certain circumstances. How can we do such a thing? It is, in fact, very easy. Program crashes can arise from many different causes. One of the more common is when the program tries to divide by zero. In mathematics, the result of taking any number and dividing it by zero is called "undefined." In a computer, "undefined" is a serious error and the program cannot continue, so it crashes. Therefore, one simple way to make a program crash deliberately is to insert a couple of extra instructions into the program that will divide a number by zero. In fact, that is exactly how I produced the TroubleMaker.exe example in the figure above.

Now we begin the main proof of the impossibility of a crash-finding program. The figure on the following page summarizes the flow of the argument. We start off assuming the existence of CanCrash.exe, which is a yes-no program that always terminates, outputting "yes" if the program it receives as input can ever crash under any circumstances, and outputting "no" if the input program never crashes.

Now we make a somewhat weird change to CanCrash.exe: instead of outputting "yes," we will make it crash instead! (As discussed above, it's easy to do this by deliberately dividing by zero.) Let's call the resulting program CanCrashWeird.exe. So this program deliberately crashes—causing the appearance of a dialog box similar to the one above—if its input can crash, and outputs "no" if its input never crashes.

The next step shown in the figure is to transform CanCrashWeird.exe into a more obscure beast called CrashOnSelf.exe. This



A sequence of four crash-detecting programs that cannot exist. The last program, AntiCrashOnSelf.exe, is obviously impossible, since it produces a contradiction when run on itself. However, each of the programs can be produced easily by a small change to the one above it (shown by the arrows). Therefore, none of the four programs can exist.

program, just like YesOnSelf.exe in the last section, is concerned only with how programs behave when given themselves as input. Specifically, CrashOnSelf.exe examines the input program it is given and deliberately crashes if that program would crash when run on itself. Otherwise, it outputs "no." Note that it's easy to produce CrashOnSelf.exe from CanCrashWeird.exe: the procedure is exactly the same as the one for transforming AlwaysYes.exe into YesOnSelf.exe, which we discussed on page 188.

The final step in the sequence of the four programs in the figure is to transform CrashOnSelf.exe into AntiCrashOnSelf.exe. This simple step just reverses the behavior of the program: so if its input crashes when run on itself, AntiCrashOnSelf.exe outputs "yes." But if the input doesn't crash when run on itself, AntiCrashOnSelf.exe deliberately crashes.

Now we've arrived at a point where we can produce a contradiction. What will AntiCrashOnSelf.exe do when given itself as input? According to its own description, it should output "yes" if it crashes (a contradiction, since it can't terminate successfully with the output "yes" if it has already crashed). And again according to its own description, AntiCrashOnSelf.exe should crash if it doesn't crash—which is also self-contradictory. We've eliminated both possible behaviors of AntiCrashOnSelf.exe, which means the program could not have existed in the first place.

Finally, we can use the chain of transformations shown in the figure on the facing page to prove that CanCrash.exe can't exist either. If it *did* exist, we could transform it, by following the arrows in the figure, into AntiCrashOnSelf.exe—but we already know AntiCrashOnSelf.exe can't exist. That's a contradiction, and, therefore, our assumption that CanCrash.exe exists must be false.

The Halting Problem and Undecidability

That concludes our tour through one of the most sophisticated and profound results in computer science. We have proved the absolute impossibility that anyone will ever write a computer program like CanCrash.exe: a program that analyzes other programs and identifies all possible bugs in those programs that might cause them to crash.

In fact, when Alan Turing, the founder of theoretical computer science, first proved a result like this in the 1930s, he wasn't concerned at all about bugs or crashes. After all, no electronic computer had even been built yet. Instead, Turing was interested in whether or not a given computer program would eventually produce an answer. A closely related question is: will a given computer program ever *terminate*—or, alternatively, will it go on computing forever, without producing an answer? This question of whether a given computer program will eventually terminate, or "halt," is known as the Halting Problem. Turing's great achievement was to prove that his variant of the Halting Problem is what computer scientists call "undecidable." An undecidable problem is one that can't be solved by writing a computer program. So another way of stating Turing's result is: you can't write a computer program called AlwaysHalts.exe, that outputs "yes" if its input always halts, and "no" otherwise.

Viewed in this way, the Halting Problem is very similar to the problem tackled in this chapter, which we might call the Crashing Problem. We proved the undecidability of the Crashing Problem, but you can use essentially the same technique to prove the Halting Problem

is also undecidable. And, as you might guess, there are many other problems in computer science that are undecidable.

WHAT ARE THE IMPLICATIONS OF IMPOSSIBLE PROGRAMS?

Except for the conclusion, this is the last chapter in the book. I included it as a deliberate counterpoint to the earlier chapters. Whereas every previous chapter championed a remarkable idea that renders computers even more powerful and useful to us humans, in this chapter we saw one of the fundamental limitations of computers. We saw there are some problems that are literally impossible to solve with a computer, regardless of how powerful the computer is or how clever its human programmer. And these undecidable problems include potentially useful tasks, such as analyzing other computer programs to find out whether they might crash.

What is the significance of this strange, and perhaps even foreboding, fact? Does the existence of undecidable problems affect the way we use computers in practice? And how about the computations that we humans do inside our brains—are those also prevented from tackling undecidable problems?

Undecidability and Computer Use

Let's first address the practical effects of undecidability on computer use. The short answer is: no, undecidability does not have much effect on the daily practice of computing. There are two reasons for this. Firstly, undecidability is concerned only with whether a computer program will ever produce an answer, and does not consider how long we have to wait for that answer. In practice, however, the issue of efficiency (in other words, how long you have to wait for the answer) is extremely important. There are plenty of decidable tasks for which no efficient algorithm is known. The most famous of these is the Traveling Salesman Problem, or TSP for short. Restated in modern terminology, the TSP goes something like this: suppose you have to fly to a large number of cities (say, 20 or 30 or 100). In what order should you visit the cities so as to incur the lowest possible total airfare? As we noted already, this problem *is* decidable—in fact, a novice programmer with only a few days' experience can write a computer program to find the cheapest route through the cities. The catch is that the program could take millions of years to complete its job. In practice, this isn't good enough. Thus, the mere fact that a problem is decidable does not mean that we can solve it in practice.

Now for the second reason that undecidability has limited practical effects: it turns out that we can often do a good job of solving undecidable problems *most of the time*. The main example of the current chapter is an excellent illustration of this. We followed an elaborate proof showing that no computer program can ever be capable of finding all the bugs in all computer programs. But we can still try to write a crash-finding program, hoping to make it find most of the bugs in most types of computer programs. This is, indeed, a very active area of research in computer science. The improvements we've seen in software reliability over the last few decades are partly due to the advances made in crash-finding programs. Thus, it is often possible to produce very useful partial solutions to undecidable problems.

Undecidability and the Brain

Does the existence of undecidable problems have implications for human thought processes? This question leads directly to the murky depths of some classic problems in philosophy, such as the definition of consciousness and the distinction between mind and brain. Nevertheless, we can be clear about one thing: if you believe that the human brain could, in principle, be simulated by a computer, then the brain is subject to the same limitations as computers. In other words, there would be problems that no human brain could solve—however intelligent or well-trained that brain might be. This conclusion follows immediately from the main result in this chapter. If the brain can be imitated by a computer program, and the brain can solve undecidable problems, then we could use a computer simulation of the brain to solve the undecidable problems also—contradicting the fact that computer programs cannot solve undecidable problems.

Of course, the question of whether we will ever be able to perform accurate computer simulations of the brain is far from settled. From a scientific point of view, there do not seem to be any fundamental barriers, since the low-level details of how chemical and electrical signals are transmitted in the brain are reasonably well understood. On the other hand, various philosophical arguments suggest that somehow the physical processes of the brain create a "mind" that is qualitatively different from any physical system that could be simulated by computer. These philosophical arguments take many forms and can be based, for example, on our own capacity for self-reflection and intuition, or an appeal to spirituality.

There is a fascinating connection here to Alan Turing's 1937 paper on undecidability—a paper that is regarded by many as the foundation of computer science as a discipline. The paper's title

is, unfortunately, rather obscure: it begins with the innocuous-sounding phrase "On computable numbers..." but ends with the jarring "...with an application to the Entscheidungsproblem." (We won't be concerning ourselves with the second part of the title here!) It is crucial to realize that in the 1930s, the word "computer" had a completely different meaning, compared to the way we use it today. For Turing, a "computer" was a *human*, doing some kind of calculation using a pencil and paper. Thus, the "computable numbers" in the title of Turing's paper are the numbers that could, in principle, be calculated by a human. But to assist his argument, Turing describes a particular type of machine (for Turing, a "machine" is what we would call a "computer" today) that can also do calculations. Part of the paper is devoted to demonstrating that certain calculations cannot be performed by these machines—this is the proof of undecidability, which we have discussed in detail already. But another part of the same paper makes a detailed and compelling argument that Turing's "machine" (read: computer) can perform any calculation done by a "computer" (read: human).

You may be beginning to appreciate why it is difficult to overstate the seminal nature of Turing's "On computable numbers..." paper. It not only defines and solves some of the most fundamental problems in computer science, but also strikes out into the heart of a philosophical minefield, making a persuasive case that human thought processes could be emulated by computers (which, remember, had not been invented yet!). In modern philosophical parlance, this notion—that all computers, and probably humans too, have equivalent computational power—is known as the *Church–Turing thesis*. The name acknowledges both Alan Turing and Alonzo Church, who (as mentioned earlier) independently discovered the existence of undecidable problems. In fact, Church published his work a few months before Turing, but Church's formulation is more abstract and does not explicitly mention computation by machines.

The debate over the validity of the Church–Turing thesis rages on. But if its strongest version holds, then our computers aren't the only ones humbled by the limits of undecidability. The same limits would apply not only to the genius at our fingertips, but the genius behind them: our own minds.

11

Conclusion: More Genius at Your Fingertips?

> We can only see a short distance ahead, but we can see plenty there that needs to be done.
>
> —ALAN TURING, *Computing Machinery and Intelligence*, 1950

I was fortunate, in 1991, to attend a public lecture by the great theoretical physicist Stephen Hawking. During the lecture, which was boldly titled "The Future of the Universe," Hawking confidently predicted that the universe would keep expanding for at least the next 10 billion years. He wryly added, "I don't expect to be around to be proved wrong." Unfortunately for me, predictions about computer science do not come with the same 10-billion-year insurance policy that is available to cosmologists. Any predictions I make may well be disproved during my own lifetime.

But that shouldn't stop us thinking about the future of the great ideas of computer science. Will the great algorithms we've explored remain "great" forever? Will some become obsolete? Will new great algorithms emerge? To address these questions, we need to think less like a cosmologist and more like a historian. This brings to mind another experience I had many years ago, watching some televised lectures by the acclaimed, if controversial, Oxford historian A. J. P. Taylor. At the end of the lecture series, Taylor directly addressed the question of whether there would ever be a third world war. He thought the answer was yes, because humans would probably "behave in the future as they have done in the past."

So let's follow A. J. P. Taylor's lead and bow to the broad sweep of history. The great algorithms described in this book arose from incidents and inventions sprinkled throughout the 20th century. It seems reasonable to assume a similar pace for the 21st century, with a major new set of algorithms coming to the fore every two or three decades. In some cases, these algorithms could be stunningly original, completely new techniques dreamed up by scientists. Public key