Computability

We write programs in order to solve problems. But if a problem is "too hard" and our program is "too simple," this won't work. The sophistication of our program needs to match the intrinsic difficulty of the problem.

Can this be made precise? Theory of computation attempts just that. We will study classes of programs, trying to characterize the classes of problems they solve. The approach is mathematical: we try to prove theorems.

In this first chapter we do not place any restrictions on the programs; later we will restrict the kinds of data structures the programs use.

Maybe surprisingly, even without any restrictions on the programs we will be able to prove in this first chapter that many problems cannot be solved.

The central piece of mathematics in this first chapter is a proof technique called "proof by contradiction." The central piece of computer science is the notion that the input to a program can be a program.

1.1 Strings, Properties, Programs

Programs often operate on other programs. Examples of such "tools" are editors, compilers, interpreters, debuggers, utilities such as — to pick some `Unix` examples — `diff`, `lint`, `rcs`, `grep`, `cat`, `more` and lots of others. Some such programs determine properties of programs they are operating on. For example, a compiler among other things determines whether or not the source code of a program contains any syntax errors. Figure 1.1 shows a much more modest example. The program "`oddlength.cpp`" determines whether or not its input consists of an odd number of characters. If we ran the program in Figure 1.1 on some text file, e.g. by issuing the two commands:

```
CC -o oddlength oddlength.cpp
oddlength < phonelist
```

the response would be one of two words, `Yes` or `No` (assuming that the program

```
void main( void )
{
    if (oddlength())
        cout << "Yes";
    else
        cout << "No";
}

boolean oddlength( void )
{
    boolean toggle = FALSE;

    while (getchar() != EOF)
    {
       toggle = !toggle;
    }
    return( toggle );
}
```

Figure 1.1: A program "`oddlength.cpp`" that determines whether or not its input consists of an odd number of characters

was stored in `oddlength.cpp` and that there was indeed a text file `phonelist` and that this all is taking place on a system on which these commands make sense[1]). We could of course run this program on the source texts of programs, even on its own source text:

```
CC -o oddlength oddlength.cpp
oddlength < oddlength.cpp
```

The response in this case would be `No` because it so happens that the source code of this program consists of an even number of characters.

Mathematically speaking, writing this program was a "constructive" proof of the following theorem. To show that there was a program as claimed by the theorem we constructed one.

Theorem 1 *There is a program that outputs* `Yes` *or* `No` *depending on whether or not its input contains an odd number of characters. (Equivalently, there is a boolean function that returns* `TRUE` *or* `FALSE` *depending on whether or not its input*[2] *contains an odd number of characters.)*

In theory of computation the word "decidable" is used to describe this situation, as in

[1] For our examples of programs we assume a Unix environment and a programming language like C or C++. To avoid clutter we leave out the usual "`#include`" and "`#define`" statements and assume that `TRUE` and `FALSE` have been defined to behave the way you would expect them to.

[2] The "input" to a function is that part of the standard input to the program that has not been read yet at the time the function gets called.

```
void main( void )
{
    char c;

    c = getchar();
    while (c != EOF)
    {
       while (c == 'X'); // loop forever if an 'X' is found
       c = getchar();
    }
}
```

Figure 1.2: A program "x.cpp" that loops on some inputs

Theorem 1 (new wording) *Membership in the set* $L_{ODDLENGTH} = \{x : |x| \text{ is odd}\}$ *is decidable.*

Properties vs. sets

The notion of a "property" is the same as the notion of a "set": x having property Π is the same as x being in the set $\{z : z \text{ has property } \Pi\}$. This suggests yet another rewording of Theorem 1 which uses standard terminology of theory of computation. If we use the name ODDLENGTH for the property of consisting of an odd number of characters then the theorem reads as follows.

Theorem 1 (yet another wording) ODDLENGTH *is decidable.*

More interesting properties of programs

Concern about whether or not a program contains an odd number of characters is not usually uppermost on programmers' minds. An example of a property of much greater concern is whether or not a program might not always terminate. Let's say a program has property LOOPING if there exists an input on which the program will not terminate. Two examples of programs with property LOOPING are shown in Figures 1.2 and 1.3. In terms of sets, having property LOOPING means being a member of the set $L_{\text{LOOPING}} = \{x : x \text{ is a syntactically correct program and there is an input } y \text{ such that } x, \text{ when run on } y, \text{ will not terminate}\}$.

Instead of being concerned with whether or not there is some input (among the infinitely many possible inputs) on which a given program loops, let us simplify matters and worry about only one input per program. Let's make that one input the (source code of the) program itself. Let SELFLOOPING be the property that a program will not terminate when run with its own source text as input. In terms of sets, SELFLOOPING means being a member of the set $L_{\text{SELFLOOPING}} = \{x : x \text{ is a syntactically correct program and, when run on } x, \text{ will not terminate}\}$.

The programs in Figures 1.2 and 1.3 both have this property. The program in Figure 1.1 does not.

The bad news about SELFLOOPING is spelled out in the following theorem.

Theorem 2 SELFLOOPING *is not decidable.*

```
void main( void )
{
    while (TRUE);
}
```

Figure 1.3: Another program, "`useless.cpp`," that loops on some (in fact, all) inputs

What this means is that there is no way to write a program which will print `"Yes"` if the input it reads is a program that has the property SELFLOOPING, and will print `"No"` otherwise. Surprising, maybe, but true and not even very hard to prove, as we will do a little later, in Section 1.3, which starts on page 12.

Strings vs. programs The definitions of LOOPING and SELFLOOPING use the notion of a "syntactically correct program." The choice of programming language does not matter for any of our discussions. Our results are not about peculiarities of somebody's favorite programming language, they are about fundamental aspects of any form of computation. In fact, to mention syntax at all is a red herring. We could do all our programming in a language, let's call it C−−, in which *every* string is a program. If the string is a C or C++ program, we compile it with our trusty old C or C++ compiler. If not, we compile it into the same object code as the program "`main () {}`." Then every string is a program and "running x on y" now makes sense for any strings x and y. It means that we store the string x in a file named, say, `x.cmm`, and the string y in a file named, say, `y`, and then we use our new C−− compiler and do

```
CC-- -o x x.cmm
x < y
```

So let's assume from now on that we are programming in "C−−," and thus avoid having to mention "syntactically correct" all the time.

One convenient extension to this language is to regard Unix shell constructs like

```
grep Boulder | sort
```

as programs. This program can be run on some input by

```
cat phonebook | grep Boulder | sort
grep Boulder phonebook | sort
```

or by

```
( grep Boulder | sort ) < phonebook
```

1.2 A Proof By Contradiction

Before we prove our first undecidability result here is an example that illustrates the technique of proving something "by contradiction."

Proving a theorem by contradiction starts with the assumption that the theorem is not true. From this assumption we derive something that we know is not true. The conclusion then is that the assumption must have been false, the theorem true. The "something" that we know is not true often takes the form of "*A and not A*."

Theorem 3 *$\sqrt{2}$ is irrational.*

(A rational number is one that is equal to the quotient of two integers. An irrational number is one that is not rational.)

Proof (Pythagoreans, 5th century B.C.) [3] For the sake of deriving a contradiction, assume that $\sqrt{2}$ is rational.

Then

$$\sqrt{2} = \frac{p}{q} \tag{1.1}$$

for two integers p and q which have no factors in common: they are "relatively prime." Squaring both sides of (1.1) yields

$$2 = \frac{p^2}{q^2} \tag{1.2}$$

and therefore

$$2 \times q^2 = p^2 \tag{1.3}$$

This shows that p^2 contains a factor of 2. But p^2 cannot have a factor of 2 unless p did. Therefore p itself must contain a factor of 2, which gives p^2 at least two factors of 2:

$$p^2 = 2 \times 2 \times r \tag{1.4}$$

for some integer r. Combining (1.3) and (1.4) yields

$$2 \times q^2 = 2 \times 2 \times r \tag{1.5}$$

which simplifies to

$$q^2 = 2 \times r \tag{1.6}$$

which shows that q also contains a factor of 2, contradicting the fact that p and q were relatively prime. □

Next we apply this proof technique to get our first undecidability result.

[3] according to Kline, *Mathematical Thought from Ancient to Modern Times*, New York, 1972, pp.32-33

```
void main( void )
{
    if (selflooping())
        ;                    // stop, don't loop
    else
        while (TRUE);        // do loop forever
}

boolean selflooping( void )  // returns TRUE if the input
{                            // is a program which, when
                             // run with its own source code
    ...                      // as input, does not terminate;
                             // FALSE otherwise
}
```

Figure 1.4: A program "`diagonal.cpp`" that would exist if SELFLOOPING were decidable