The "Affirmative" Problem

This note is about the limits of computation; about something computers cannot do. To keep things simple, it only address a simple kind of program: the kind that is programmed to read an input and output a boolean. (It is traditional to call the boolean outputs "accept" and "reject", but we will usually call then "true" and "false" instead).

Let's say M is a program that takes an input x and has two possible outputs "true" and "false". Running M on input x therefore has three possible outcomes: M halts and outputs "true", M halts and outputs "false", or M never halts because it's in an infinite loop (and therefore outputs nothing). The interesting thing about the third possibility is that, in general, it is hard to tell whether a program is in an infinite loop or just taking a very long time to complete.

There is a famous theorem in computer science that says: There is no algorithm that can take an arbitrary program M and input x and determine whether M halts on x. This is called the "Halting Problem". Clearly, for each (M,x) there must be an answer: either M halts when given x (given enough time) or it does not.

Note that for some programs and inputs you can easily tell that they halt. But, what the Halting Problem says is there is no general solution that for *every* (M,x) can determine "does M halt on x".

This is likely the first time that you've encountered a simple sounding problem that a computer is unable to solve.

Here's a related theorem that forms the basis of the halting problem. We'll look at the connection between the two problems in the next set of notes.

Theorem: There is no algorithm capable of determining whether an arbitrary program eventually outputs "true" for an arbitrary input.

This type of theorem is usually proven by contradiction. We'll assume that such an algorithm exists and then show that a logical impossibility results. That will mean that the assumption was false and therefore no such algorithm exists.

Let's say that *Affirmative* is a program and that for any (M,x), *Affirmative*(M,x) outputs "true" if M(x) eventually outputs "true" and *Affirmative*(M,x) outputs "false" if M(x) either eventually outputs "false" or never halts. So, *Affirmative* always halts, even if M sometimes doesn't.

Now consider a new program D that calls *Affirmative* as a subroutine:

```
D(M):
    if Affirmative(M,M) outputs "true"
        output "false"  // output "false" if M outputs "true" for M
    else
        output "true"   // output "true" if M does not output "true" for M
```

Because we're assuming *Affirmative* is well defined, D is too. So, D takes a program M as input and then passes that program to *Affirmative* both as the program M that *Affirmative* analyzes but also as the input that M is analyzed for. This sounds odd, but is plausible. If the way we represent M is as M's source code, then if M were a source code compiler, then we'd be supplying a source code to a compiler, which makes perfect sense. Finally, if *Affirmative* determines that M outputs "true" on input M, then D outputs "false", otherwise D outputs "true".

Here's where things get really weird. What happens if we run D(D)? By definition (just substituting D for M in the above definition),

```
D(D):
    if Affirmative(D,D) outputs "true"
        output "false"   // output "false" if D outputs "true" for D
    else
        output "true"    // output "true" if D does not output "true" for D
```

Using this construction, we can prove by contradiction that an *Affirmative* algorithm cannot work on all inputs. Let's assume *Affirmative* exists and works on all inputs. Let's look at two cases.

Case 1. Let's say *Affirmative*(D,D) returns "true". By the definition of *Affirmative* this means D(D) must return "true". But, in fact if you look at the definition of D(D) when *Affirmative*(D,D) outputs "true", D(D) outputs "false". This is a contradiction because D(D) cannot return "true" and "false" simultaneously.

Case 2. Let's say *Affirmative*(D,D) returns "false". By the definition of *Affirmative* this means D(D) does not return "true". But, in fact if you look at the definition of D(D) when *Affirmative*(D,D) outputs "false", D(D) outputs "true". This is a contradiction because D(D) cannot return "true" and not return "true" simultaneously.

So, if an *Affirmative* algorithm existed that worked on all inputs it would always lead to one of the above contradictions. Therefore an *Affirmative* algorithm must not exist.

Again, this proves that there is no *Affirmative* algorithm that works on all (M,x). It is possible that an *Affirmative* algorithm exists that works on most (M,x), but not one that works on all (M,x).