The Halting Problem

We define the Halting Problem for Turing machines and prove that it is unsolvable by a Turing machine, and therefore unsolvable by any algorithm (by the Church-Turing thesis.)

Statement of the Halting Problem

We consider the Turing machine with the instructions:

```
((q1, b, q1, 1, R)
 (q1, 0, q2, 0, L)
 (q1, 1, q1, 1, R))
```

If we start this machine on any finite string of 1's, it moves right forever writing 1's as it goes – it does not halt. If instead we start this machine on a string with 0's and 1's, it halts when it reads the first 0. For example:

```
-------------------------------------------------------------
  |     |  1  |  1  |  0  |  1  |     |     |     |     |
-------------------------------------------------------------
           ^
           q1
-------------------------------------------------------------
  |     |  1  |  1  |  0  |  1  |     |     |     |     |
-------------------------------------------------------------
                 ^
                 q1
-------------------------------------------------------------
  |     |  1  |  1  |  0  |  1  |     |     |     |     |
-------------------------------------------------------------
                       ^
                       q1
-------------------------------------------------------------
  |     |  1  |  1  |  0  |  1  |     |     |     |     |
-------------------------------------------------------------
                 ^
                 q2
```

It would be convenient to have an algorithm to determine whether a Turing machine will halt or run forever for a given input tape. This leads to the following question:

```
The Halting Problem:

Given a Turing machine T and string x, does T eventually halt
when started in state q1 with its head on the leftmost
symbol of x and the rest of the tape blank?
```

To formalize this as a computational problem to be solved by a Turing machine, we have to specify the representation of inputs and outputs as strings of symbols. For concreteness, we will assume that the Turing machine `T` has tape symbols blank, 0, and 1. The input `x` will just be a string of 0's and 1's.

We represent T as a string of 0's and 1's by first removing the (redundant) `q` symbols and translating each remaining character of its representation into a 4-bit code as follows.

```
b = 0000
0 = 0001
1 = 0010
( = 0011
) = 0100
, = 0101
2 = 0110
3 = 0111
4 = 1000
5 = 1001
6 = 1010
7 = 1011
8 = 1100
9 = 1101
L = 1110
R = 1111
```

The translation of the instructions listed for the Turing machine above would begin:

```
(    (    1    ,    b    ,    1    ,    1    ,    R    )    (    ...
0011 0011 0010 0101 0000 0101 0010 0101 0010 0101 1111 0100 0011 ...
```

For visual assistance, I have put blanks between the 4-bit codes, but you should imagine the result as a string `t` of 140 0's and 1's.

For a Turing machine `H` to solve the Halting Problem with this input representation would mean that if we start `H` on the leftmost symbol of a string `z` of 0's and 1's, `H` eventually halts with just one nonblank symbol (either 0 or 1) on the tape and its head on that symbol. The output symbol should be 1 if in the representation above, the initial portion of `z` represents a Turing machine `T` and the rest of `z` is a string `x`, and `T` eventually halts when started on `x`. Otherwise, the output symbol on the tape when `H` halts should be 0. (This also covers the case when the string `z` does not have the correct syntax to represent a Turing machine `T` and input string `x`.)

Recall that `t` is the string that represents the Turing machine in the example above. As examples of the values that should be computed we have the following. The first one represents the example Turing machine with input `111` and the second one the example Turing machine with input `1101`.

```
t111  =>  0
t1101 =>  1
```

We show that the Halting Problem is algorithmically unsolvable. That is, there is no program that always halts and correctly answers 1 or 0 for every possible pair of inputs `T` and `x`. Specifically, we show that no Turing machine can solve this problem, and apply the Church-Turing thesis to conclude that it is algorithmically unsolvable. We also use the term undecidable for algorithmically unsolvable yes/no questions like this one.

The Proof

How do we show that the Halting Problem cannot be solved by a Turing machine? We use a proof by contradiction: we assume to the contrary that there is a Turing machine `H` to solve the Halting Problem, and show that this leads to a contradiction. We assume `H` exists and has the following property:

```
For any inputs T and x,
H halts and outputs 1 if T halts on input x,
and H halts and outputs 0 otherwise.
```

Note that `H` is required to halt in both cases. We construct a new Turing machine `Q` consisting of the instructions of `H` with two new instructions added. Without loss of generality we may require that state `q2` is the halt state for machine `H` when it outputs a 1. (If it isn't, we can renumber the existing states so that this condition holds.) That is, in this case, `H` halts because there is no instruction `(q2, 1, ...)`. In a similar way, we can require that state `q3` does not appear in the instructions of `H`. Then, to construct `Q`, we add to the instructions of `H` the following two instructions:

```
(q2, 1, q3, 1, R)
(q3, b, q3, b, R)
```

The effect of these two instructions is to cause `Q` to run forever on those inputs `z` on which `H` halts with output 1. For those inputs `z` for which `H` halts with output 0, `Q` also halts with output 0. Thus, the behavior of `Q` is as follows:

```
For inputs on which H halts with output 1, Q doesn't halt.
For inputs on which H halts with output 0, Q halts with output 0.
```

Now we create a new Turing machine, `C` that takes an input string `x` and makes a copy of the string `x` immediately to the right of the string `x`, moves to the leftmost symbol of the string `x`, and then transfers to the initial state of `Q`, (where the states of `Q` have been renumbered to be different from the other states of `C`.) The copying operation of `C` is similar to the copying machine described in an earlier lecture, but without the symbol `c`.

(Referring to the proof of the unsolvability of the Halting Problem for Scheme programs in Lecture 10, the Turing machine `H` corresponds to the Scheme `halts?` procedure and the Turing machine `C` corresponds to the Scheme `contrary` procedure.)

Then `C` is a Turing machine, represented by the list of its instructions. Translate those instructions into a string of 0's and 1's as above, and call the resulting string `y`. What happens when we run `C` on input `y`, that is, run the machine `C` on the representation of its own instructions? It makes a copy of its input and calls `Q` on the result, that is, it calls `Q` on the input `yy`. Thus:

```
The result of running C on input y is the same
as the result of running Q on input yy.
```

And what is the result of running `Q` on input `yy`? Recall the relation between `Q` and `H`:

```
Q does not halt on input yy if H halts with output 1 on yy.
Q halts with output 0 on input yy if H halts with output 0 on yy.
```

So, what is the result of running `H` on input `yy`? Because `yy` correctly represents the Turing machine `C` and input `y`, and `H` solves the Halting Problem, we have that

```
H halts with output 1 if Turing machine C halts on input y.
H halts with output 0 if Turing machine C doesn't halt on input y.
```

Thus, working backwards, if Turing machine `C` halts on input `y` then `H` halts with output 1 on input `yy`, and `Q` does not halt on input `yy`, so `C` does not halt on input `y`. In other words, if `C` halts on input `y`, then `C` does not halt on input `y`.

However, if `C` does not halt on input `y`, then, again working backwards, `H` halts with output 0 on input `yy`, and therefore `Q` halts with output 0 on input `yy`, so `C` with input `y` halts with output 0. In other words, if `C` does not halt on input `y`, then `C` halts on input `y`.

This completes the contradiction, because we've shown that `C` halts on input `y` if and only if it doesn't halt on input `y`. Thus, the Turing machine `H` cannot exist, that is, the Halting Problem is algorithmically unsolvable.

Lest you think that this is just nonsense incurred by the folly of running a program on its own instructions, consider the benefits of having a compiler compile its own code, or a C syntax-checker written in C run on its own code as a test case.