Notes: 3-21-16 (The Halting Problem)

I. Building up to the Halting Problem

A. The idea of a “Universal Turing Machine”

1. A universal Turing Machine is some Turing Machine that is able to take as input another Turing Machine and some other input to simulate on it.

2. Formally, a Turing Machine of this form accepts input that looks like this:

\(<M, w>\)

a. \(M\) = description of a Turing Machine
b. \(w\) = input string that we run \(M\) on

3. We can use a Turing Machine that takes other Turing Machines as input so long as we have some agreed-upon convention for describing Turing Machines as character strings. Here’s an example of such a convention:

Ex: Say we have this situation below.

- We need to specify 5 things if we are to put the transitions that occur in a Turing Machine into a string:

  i. The state we are currently at (designated as state \(i\))
  ii. The character we read (call it \(k\))
  iii. The direction we move when we read \(k\) (denoted as 0 or 1 for left or right, defined with \(m\))
  iv. What we write after we read \(k\) (call it \(l\))
  v. The state we end up at (designate as state \(j\))

- Now we can make a code out of this transition of form:

  \(0101010010101\)

- Where every bit of information is specified by some string of 0’s and separated by a single 1. This defines a single code, and to create a series of transitions that define a Turing Machine, separate every coded transition by a pair of 1’s.

  11code11code11code...
- And now we have our convention for describing Turing Machines as character strings. Remember that it's just a single convention; there are multiple ways of creating conventions, but for the sake of the notes, we are sticking to this particular convention.

4. With the convention established, we can now prove that the universal Turing Machine can indeed be run.

*Formal Theorem:* Let \( L_u = \{<M, w> | w \in L(M) \text{ or that } w \text{ is accepted by } M \} \)

*Formal Proof:* Do the following steps:

i. Check the format of \( M \). Does it follow the convention for Turing Machines? If so, continue to step ii.

ii. Put the code for \( M \) on the tape 1 for our universal Turing Machine. Put \( w \) on tape 2 and a 0 on tape 3.

iii. While tape 3's character string is not equal to 00, do the following:

   a. Let \( 0^i \) = contents of tape 3. Also let \( x_k \) = the symbol we're reading on tape 2.

   b. Scan tape 1 for the prefix: \( 110^i10^k \). (Remember the convention we're using.) Two cases arise:

      - The prefix is not in tape 1. Reject the input.
      - The prefix is in tape 1. Put a new state \( 0^i \) on tape 3. Write \( 0^i \) on tape 2 and move on tape 2 left or right.

B. Why you can’t make a table for all Turing Machines and Inputs

1. Let's start by attempting to make a table that takes all possible Turing Machines and all possible inputs we could give them. Assume that \( M_i \) is a Turing Machine and \( w_j \) is a possible string, making any location in the table of form \((i, j)\).

<table>
<thead>
<tr>
<th></th>
<th>( w_0 )</th>
<th>( w_1 )</th>
<th>( w_2 )</th>
<th>( w_3 )</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>( M_0 )</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>( M_1 )</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>( M_2 )</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>( M_3 )</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
2. For any \((i, j)\), \(M_i\) will accept string \(w_j\) if the value of \((i, j)\) is 1 and reject it if the value of \((i, j)\) is 0.

3. Now, take the diagonal that goes from the top left of the table to the bottom right of it. Take all those values and complement them. Let \(D\) = the complement of the diagonal, where for any entry \((i, j)\), \(w_j\) is not accepted by \(M_i\). This is succinctly represented by \(L_D = \{w_j \mid w_j \text{ is not accepted by } M_i\}\).

4. Remember that this table we constructed is supposed to represent EVERY possible result for EVERY Turing Machine for EVERY input. Yet the existence of \(D\) defies this completely, since it presents strings that aren’t accepted by ANY Turing Machine. The table is incomplete, and because the table is supposedly infinite, it will never be complete. Thus, making such a table is fruitless.

5. If you’re still not convinced, here’s a brief proof by contradiction:
   a. Assume \(L_D = L(M_i)\) for some \(i\).
   b. You have two cases:
      i. \(w_i\) is in \(L(M_i)\). This implies that \(w_i\) is not in \(L_D\), but this would then imply that \(w_i\) is NOT in \(L(M_i)\).
      ii. \(w_i\) is not in \(L(M_i)\). This implies that \(w_i\) is in \(L_D\), but this would then imply that \(w_i\) IS in \(L(M_i)\).
   c. In either case, you create a contradiction, and thus \(L_D\) is not in the language of any Turing Machine.

C. You will need both of these concepts to properly understand the Halting Problem. Please review them if you are uncertain of them before you continue reading.

II. The Halting Problem—and its Applications

A. The Halting Problem
   1. Before the 1930’s, people believed that every problem was computable, given infinite time and resources.
   2. This was disproven in 1931, when Kurt Gödel presented his incompleteness theorem. This has much to what we worked with in section I.B. In short, there are some problems that just cannot be solved; they are referred to as undecidable.
   3. The Halting Problem is a basic undecidable problem. It goes like this:
a. “Suppose that we have a Turing Machine M and an input w. Will M accept w (will M halt, not running forever on w)?”
b. The challenge is whether you can determine if M will accept w (not the result of acceptance or not in of itself, but whether you can actually obtain an answer in the first place).

4. Long story short, it’s impossible to get an answer to the Halting Problem. Prove this by contradiction:
   a. Assume that the Halting Problem can be solved. That means there exists a Turing Machine, given any description of another Turing Machine and an input w to simulate on it, that can reliably answer whether the machine passed as input will accept w or not all of the time.
   b. However, if such a machine exists, then it should also be able to accept $L_0$ (the same $L_0$ from section I.B.3), which we already established cannot be accepted by any Turing Machine that exists... Which would include this hypothetical machine. This contradiction proves that no such machine exists and the Halting Problem is unsolvable.

B. Applying the Logic of the Halting Problem

1. So we know that the Halting Problem is unsolvable. What’s the big deal? The big deal is that we can use that fact to prove that other problems are undecidable as well.

2. Let’s walk through an example. Say you’re given this: “prove that given a Turing Machine M, it is undecidable if $L(M) = \emptyset$ (the language that accepts M is empty).”
   Prove this by contradiction:
   a. Assume we can decide if $L(M) = \emptyset$. Then there exists a Turing Machine T such that T(M) is equal to “yes” if $L(M) = \emptyset$ and “no” if $L(M) \neq \emptyset$. With this, we can then have T take $<M, w>$ as input and determine whether M accepts w or not.
   b. To determine whether M accept w, construct some new Turing Machine M’ that ignores its input and simply runs $<M, w>$. If $<M, w>$ halts, M’ accepts the input. If $<M, w>$ does not halt, M’ rejects the input.
      i. $L(M’)$, then, must either accept every string... Or accept none of the strings, since M’ does not care for the input itself, only whether $<M, w>$ halts or not.
c. Now give \( T <M', w> \) as input. \( T(M') \) will equal “yes” if \( L(M') \neq \emptyset \) iff \( M(w) \) halts, and \( T(M') \) will equal “no” if \( L(M') = \emptyset \) iff \( M(w) \) does not halt.

d. It looks like the problem’s solved... But pause to think for a moment. We just created a machine that determines whether another machine will halt or not given a certain input. In effect, we’ve just solved the Halting Problem, which we know for a fact that it is undecidable. Because of this contradiction, \( T \) cannot exist, and it’s undecidable whether \( L(M) = \emptyset \).

3. This type of problem solving is complicated. It all stems back to the original machine, and you need to be able to construct multiple machines to show that they solve an unsolvable problem.