Solvability and the Halting Problem.

Our development period is over. Now it is time for some action. We have the tools and materials and we need to get to work and discover some things that are not computable. We know they are there and now it is time to find and examine a few.

Our task in this section is to find some noncomputable problems. However we must first discuss what exactly problems are. Many of our computational tasks involve questions or decisions. We shall call these problems. For example, some problems involving numbers are:

- Is this integer a prime?
- Does this equation have a root between 0 and 1?
- Is this integer a perfect square?
- Does this series converge?
- Is this sequence of numbers sorted?

As computer scientists, we are very aware that not all problems involve numbers. Many of the problems that we wish to solve deal with the programs we write. Often we would like to know the answers to questions concerning our methods, or our programs. Some of these problems or questions are:

- Is this program correct?
- How long will this program run?
- Does this program contain an infinite loop?
- Is this program more efficient than that one?

A brief side trip to set forth more definitions and concepts is in order. We must describe some other things closely related to problems or questions. In fact, often when we describe problems we state them in terms of relations or predicates. For example, the predicate Prime(x) that indicates prime numbers could be defined:

Prime(x) is true if and only if x is a prime number.

and this predicate could be used to define the set of primes:

PRIMES = { x | Prime(x) }.
Another way to link the set of primes with the predicate for being a prime is to state:

\[ x \in \text{PRIMES} \text{ if and only if } \text{Prime}(x) \]

(N.B. Two comments on notation are necessary. We shall use iff to mean if and only if and will often just mention a predicate as we did above rather than stating that it is true.)

We now have several different terms for problems or questions. And we know that they are closely related. Sets, predicates, and problems can be used to ask the same question. Here are three equivalent questions:

- Is \( x \in \text{PRIMES} \)?
- Is \( \text{Prime}(x) \) true?
- Is \( x \) a prime number?

When we can completely determine the answer to a problem, the value of a predicate, or membership in a set for all instances of the problem, predicate, or things that may be in the set; we say that the problem, predicate, or set is decidable or solvable. In computational terms this means that there is a Turing machine which can in every case determine the answer to the appropriate question. The formal definition of solvability for problems follows.

Definition. A problem \( P \) is solvable if and only if there is a Turing machine \( M_i \) such that for all \( x \):

\[
M_i(x) = \begin{cases} 
1 & \text{if } P(x) \text{ is true} \\
0 & \text{if } P(x) \text{ is false}
\end{cases}
\]

If we can always solve a problem by carrying out a computation it is a solvable problem. Many examples of solvable problems are quite familiar to us. In fact, most of the problems we attempt to solve by executing computer programs are solvable. Of course, this is good because it guarantees that if our programs are correct, then they will provide us with solutions! We can determine whether numbers are prime, find shortest paths in graphs, and many other things because these are solvable problems. There are lots and lots of them. But there must be some problems that are not solvable because we proved that there are things which Turing machines (or programs) cannot do. Let us begin by formulating and examining a historically famous one.

Suppose we took the Turing machine \( M_1 \) and ran it with its own index as input. That is, we examined the computation of \( M_1(1) \). What happens? Well, in this
case we know the answer because we remember that \( M_1 \) was merely the machine:

\[
\begin{array}{c|c|c}
0 & 0 & \text{halt} \\
\end{array}
\]

and we know that it only halts when it receives an input that begins with a zero. This is fine. But, how about \( M_2(2) \)? We could look at that also. This is easy; in fact, there is almost nothing to it. Then we could go on to \( M_3(3) \). And so forth. In general, let us take some arbitrary integer \( i \) and ask about the behavior of \( M_i(i) \). And, let's not ask for much, we could put forth a very simple question: does it halt?

Let us ponder this a while. Could we write a program or design a Turing machine that receives \( i \) as input and determines whether or not \( M_i(i) \) halts? We might design a machine like the universal Turing machine that first produced the description of \( M_i \) and then simulated its operation on the input \( i \). This however, does not accomplish the task we set forth above. The reason is because though we would always know if it halted, if it went into an infinite loop we might just sit there and wait forever without knowing what was happening in the computation.

Here is a theorem about this that is very reminiscent of the result where we showed that there are more sets than computable sets.

**Theorem 1.** Whether or not a Turing machine halts when given its own index as input is unsolvable.

**Proof.** We begin by assuming that we can decide whether or not a Turing machine halts when given its own index as input. We assume that the problem is solvable. This means that there is a Turing machine that can solve this problem. Let's call this machine \( M_k \) and note that for all inputs \( i \):

\[
M_k(x) = \begin{cases} 
1 & \text{if } M_x(x) \text{ halts} \\
0 & \text{if } M_x(x) \text{ diverges} 
\end{cases}
\]

(This assertion came straight from the definition of solvability.)

Since the machine \( M_k \) exists, we can use it in the definition of another computing procedure. Consider the following machine.
This is not too difficult to construct from $M_k$ and our universal Turing machine $M_u$. We just run $M_k(x)$ until it provides an output and then either halt or enter an infinite loop.

We shall apply Church's thesis once more. Since we have developed an algorithm for the above machine $M$, we may state that is indeed a Turing machine and as such has an index in our standard enumeration. Let the integer $d$ be its index. Now we inquire about the computation of $M_d(d)$. This inquiry provides the following sequence of conclusions. (Recall that iff stands for if and only if.)

$$M_d(d) \text{ halts} \iff M(d) \text{ halts} \iff M_k(d) = 0 \iff M_d(d) \text{ diverges} \quad \text{(since $M_d = M$)} \quad \text{(see definition of $M$)} \quad \text{(see definition of $M_k$)}$$

Each step in the above deduction follows from definitions stated previously. Thus they all must be true. But there is a slight problem since a contradiction was proven! Thus something must be wrong and the only thing that could be incorrect must be some assumption we made. We only made one, namely our original assumption that the problem was solvable. This means that whether a Turing machine halts on its own index is unsolvable and we have proven the theorem.

Now we have seen an unsolvable problem. Maybe it is not too exciting, but it is unsolvable nevertheless. If we turn it into a set we shall then have a set in which membership is undecidable. This set is named $K$ and is well-known and greatly cherished by recursion theorists. It is:

$$K = \{ i \mid M_i(i) \text{ halts} \}$$

$K$ was one of the first sets to be proven undecidable and thus of great historical interest. It will also prove quite useful in later proofs. Another way to state our last theorem is:

**Corollary.** Membership in $K$ is unsolvable.

Let us quickly follow up on this unsolvability result and prove a more general one. This is possibly the most famous unsolvable problem that exists. It is called the halting problem or membership problem.
Theorem 2 (Halting Problem). For arbitrary integers i and x, whether or not $M_i(x)$ halts is unsolvable.

Proof. This follows directly from the previous theorem. Suppose we could solve halting for $M_i(x)$ on any values of i and x. All we have to do is plug in the value i for x and we are now looking at whether $M_i(i)$ halts. We know from the last theorem that this is not solvable. So the general halting problem (does $M_i(x)$ halt?) must be unsolvable also, since if it were solvable we could solve the restricted version of the halting problem, namely membership in the set K.

This is interesting from the point of view of a computer scientist. It means that no program can ever predict the halting of all other programs. Thus we shall never be able to design routines which unfailingly check for infinite loops and warn the programmer, nor can we add routines to operating systems or compilers which always detect looping. This is why one never sees worthwhile infinite loop checkers in the software market.

Let's try another problem. It seems that we cannot tell if a machine will halt on arbitrary inputs. Maybe the strange inputs (such as the machine's own index) are causing the problem. This might be especially true if we are looking at weird machines that halt when others do not and so forth! It might be easier to ask if a machine always halts. After all, this is a quality we desire in our computer programs. Unfortunately that is unsolvable too.

Theorem 3. Whether or not an arbitrary Turing machine halts for all inputs is an unsolvable problem.

Proof. Our strategy for this proof will be to tie this problem to a problem that we know is unsolvable. Thus it is much like the last proof. We shall show that halting on one's index is solvable if and only if halting for all inputs is solvable. Then since whether a machine halts on its own index is unsolvable, the problem of whether a machine halts for all inputs must be unsolvable also.

In order to explore this, let's take an arbitrary machine $M_i$ and construct another Turing machine $M_{all}$ such that:

$M_{all}$ halts for all inputs iff $M_i(i)$ halts

At this point let us not worry about how we build $M_{all}$, this will come later.

We now claim that if we can decide whether or not a machine halts for all inputs, we can solve the problem of whether a machine halts on its own index. Here is how we do it. To decide if $M_i(i)$ halts, just ask whether $M_{all}$
halts on all inputs. But, since we have shown that we cannot decide if a
machine halts upon its own index this means that if we are able to
construct $M_{\text{all}}$, then we have solved membership in $K$ and proven a
contradiction. Thus the problem of detecting halting on all inputs must
be unsolvable also.

Let us get to work. A machine like the above $M_{\text{all}}$ must be built from $M_i$.
We shall use all of our tools in this construction. As a start, consider:

\[ M(x, i) = M_u(i, i) = M_i(i) \]

Note that $M$ does not pay attention to its input $x$. It just turns the
universal machine $M_u$ loose on the input pair $(i, i)$, which is the same as
running $M_i$ on its own index. So, no matter what $x$ equals, $M$ just
computes $M_i(i)$. Yet another appeal to Church’s thesis assures us that $M$
is indeed a Turing machine and exists in the standard enumeration. Let us
say that $M$ is machine $M_m$. Thus for all $i$ and $x$:

\[ M_m(x, i) = M(x, i) = M_u(i, i) = M_i(i). \]

Now we shall call upon the s-m-n theorem. It says that there is a function
$s(m, i)$ such that for all $i$, $a$, and $x$:

\[ M_s(m, i)(x) = M_m(x, i) = M(x, i) \]

If we let all $= s(m, i)$ then we know that for fixed $i$ and all $x$:

\[ M_{\text{all}}(x) = M(x, i) = M_u(i, i) = M_i(i) \]

Another way to depict the operation of $M_{\text{all}}$ is:

\[ M_{\text{all}}(x) = \begin{cases} 
\text{halt if } M_i(i) \text{ halts} \\
\text{diverge if } M_i(i) \text{ diverges}
\end{cases} \]

To sum up, from an arbitrary machine $M_i$ we have constructed a machine
$M_{\text{all}}$ which will halt on all inputs if and only if $M_i(i)$ halts. The following
derivation shows this.

\[ M_i(i) \text{ halts} \iff M_u(i, i) \text{ halts} \]
\[ \quad \iff \text{for all } x, M(x, i) \text{ halts} \]
\[ \quad \iff \text{for all } x, M_m(x, i) \text{ halts} \]
\[ \quad \iff \text{for all } x, M_s(m, i)(x) \text{ halts} \]
\[ \quad \iff \text{for all } x, M_{\text{all}}(x) \text{ halts} \]
Each line in the above sequence follows from definitions made above or theorems (s-m-n and universal Turing machine theorems) we have proven before.

Now we have exactly what we were looking for, a machine $M_{all}$ which halts for all inputs if and only if $M_i(i)$ halts. Recalling the discussion at the beginning of the proof, we realize that our theorem has been proven.

Let us reflect on what we have done in this section. Our major accomplishment was to present an unsolvable problem. And, in addition, we presented two more which were related to it. They all concerned halting and as such are relevant to programming and computer science. From this we know that we can never get general answers to questions such as:

- will this program halt on this data set?
- will this program halt on any data set?

This is indeed a very fine state of affairs! We have shown that there is no way to ever do automatic, general checks on loops or even correctness for the programs we develop. It is unfortunate to close on such a sad note, but the actual situation is even worse! We shall presently find out that hardly anything interesting is solvable.