1 Introduction

In CSC 280 we will be formalizing computation, i.e. we will be creating precise mathematical models for describing computation. A large part of the course will involve developing these models. We will start out with extremely simple mathematical “machines” (finite automata) and work up to much more complex ones (Turing machines). Automata theory is the branch of CS theory which explores mathematical models of machines.

Once we have made computation mathematically precise, we can prove various facts about what can and cannot be computed. One of the surprising facts that arises from this endeavor is that some problems are undecidable, which means roughly that you can’t make a computer program to solve those problems. No matter how big, fast, or smart computers get, they will never be able to solve these undecidable problems. The branch of CS which studies which problems can be computed is, unsurprisingly, called computability theory.

Of the problems which can be computed, some are easier to solve than others. Harder problems require more time and space to solve. Complexity theory is the branch of CS which studies how easily different problems can be solved.

In CSC 280 you will be introduced to these three branches of computer science theory.

2 Undecidability and Diagonalization

As noted in the previous section, some problems are undecidable. Here, we give a proof that undecidable problems exist. In the next section, we’ll give an example of a famous undecidable problem known as the Halting Problem. The proof that undecidable problems exist relies on the concept of countable vs. uncountable infinite sets.

2.1 Countable and Uncountable Sets

Suppose you want to determine whether two sets contain the same number of elements. For finite sets, this is easy - you can just count everything in the first set, then count everything in the second set, and see if those two numbers are equal. But this doesn’t work for infinite sets, and so we have to develop a new way to compare the size of sets.

We can do this by pairing up elements from each set. If every element in the first set corresponds to exactly one element in the second (mathematically we call this a bijection), then the two sets have the same number of elements. This is obvious for finite sets: given sets $A = \{a_1, \ldots, a_n\}$ and $B = \{b_1, \ldots, b_n\}$, you can assign each element from $B$ to a single element of $A$. 
It is equally intuitive that the set of positive integers is the same size as the set of negative integers.

\[
\begin{array}{c|c}
A & B \\
\hline
a_1 & b_1 \\
\vdots & \\
\vdots & \\
a_n & b_n \\
\end{array}
\]

This bijection corresponds to the function \( f(x) = -x \).

However, there are also some completely counterintuitive results. For instance, it turns out that there are just as many positive even integers as positive integers. This seems strange, since the first is a strict subset of the second. But the bijection \( f(x) = 2x \) pairs every element from the positive integers with exactly one element from the even positive integers:

\[
\begin{array}{c|c}
\text{All} & \text{Even} \\
\hline
1 & 2 \\
2 & 4 \\
3 & 6 \\
\vdots & \\
\end{array}
\]

Even stranger, it turns out that the set of rational numbers has the same size as the set of integers, even though there are infinitely many rational numbers between every pair of integers.

If the rationals have the same cardinality as the positive integers, we might start to think that all infinite sets are the same size. But it turns out that this isn’t the case. It can be proven that there is no bijection from the positive integers to the real numbers. We call all infinite sets which have a bijection to the positive integers \textbf{countable}; we call those which do not \textbf{uncountable}.

We’ll prove that there are some problems that can’t be solved by a program by showing that there are uncountably many problems, but only countably many programs. First, however, we need to determine a mathematically precise definition of “problem” and “program”. We will represent programs as unique binary strings, kind of like byte code after a high-level program has been compiled. To represent problems, we need to define formal languages.

### 2.2 Formal Languages

A \textbf{formal language} is just a set of strings. These strings, like the strings you’ve seen in Java and C, are finite sequences of characters. The characters are taken from some alphabet, which we denote \( \Sigma \).
If a formal language contains only finitely many strings, we can describe the language by listing every string. Alternatively, for languages with an infinite number of strings, we can describe the language by explaining which strings it contains.

Here are some examples of formal languages over the alphabet $\Sigma = \{0, 1\}$:

- $\{0, 1, 01, 11, 00\}$
- The set of all strings with an even number of 0s.
- The set of all strings whose first character is 1.

In 173, you saw that some formal languages can be represented using regular expressions. For instance, the set of all strings whose first character is 1 could be written as $1\Sigma^*$. This same language could also be recognized by a finite automaton.

In CS theory, we like to represent all problems as formal languages, although most of these formal languages require more complicated descriptions than regular expressions can provide. Similarly, we like to represent all programs as machines which accept some formal language (and reject any string not in that formal language), although most of these machines are more complicated than finite automata. If all formal languages were decidable by some machine (i.e. if all problems could be solved by some computer program), then there would exist some bijection between formal languages and programs. We will prove that such a bijection cannot exist.

### 2.3 Diagonalization Proof that Some Formal Languages are Undecidable

First we will show that there are a countably infinite number of programs. Then we will prove that there are an uncountably infinite number of formal languages. The proofs given here are rather informal.

**Theorem:** The set of all programs is countably infinite.

**Proof:** There are infinitely many programs, but each program has finite length, and there are only finitely many programs of each length. Thus, to create a bijection between programs and positive integers, we can first list all the programs of length 1, then list all the programs of length 2, and so on.

**Theorem:** The set of all formal languages is uncountably infinite.

**Proof:** First note that the set of all finite strings is countably infinite, for the same reason that the set of all programs is countably infinite. Thus, we can list all the finite strings in an infinite list:

0 1 0 0 1 0 1 1 0 0 0 . . .

We can describe a formal language using an infinite binary string. The $n$th bit of the string will be 1 if the language contains the $n$th string in the list above, and 0 if it does not. The language containing all strings whose first character is 1 would look like this:

0 1 0 0 1 0 1 1 0 0 0 . . .

If the set of all formal languages were countably infinite, then we could list all of the formal languages in a table, like this:
To show that it is in fact uncountably infinite, we need to construct a formal language that can’t possibly be in this table. This formal language will need to differ from each formal language in the table in at least one place. We accomplish this by making a new language, where the \( n \)th bit is the opposite of the \( n \)th bit of the \( n \)th language in the table:

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>00</th>
<th>01</th>
<th>10</th>
<th>11</th>
<th>000</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Here, the red row at the bottom is the language that cannot possibly be contained in the table.

Thus it is impossible to create a bijection between the positive integers and the set of all formal languages, so the set of all formal languages is uncountable.

The technique used to prove this is called **diagonalization**, and it was first used by Cantor to show that the real numbers are uncountable. In fact, Cantor was the one who discovered uncountably infinite sets.

Because there are countably many programs, and uncountably many formal languages, there must be some formal languages which cannot be recognized by any program. These languages are called **undecidable**.

### 3 Example: The Halting Problem

The most famous example of an undecidable problem is the **Halting Problem**; its undecidability was proven by Turing.

Consider the formal language \( A_{TM} = \{ < M, x > | M \text{ halts on } x \} \). Here, \(< M, x >\) is the binary representation of a machine \( M \), followed by \( x \), a binary string. The string \(< M, x >\) is in the language if and only if machine \( M \) halts when run on \( x \).

We will prove by contradiction that \( A_{TM} \) is undecidable. First we assume that \( A_{TM} \) is in fact decidable. If this is the case, then we can build a machine \( H \) that decides \( A_{TM} \). \( H \) will return “true” if \( M \) halts on \( x \), and “false” otherwise.

Now construct a second machine, \( D \), which takes as input a machine \(< M >\) and works like this:

- Run \( H \) on \(< M, < M >>\)
- If \( H \) returns true, loop forever.
- If \( H \) returns false, halt and return true.
Basically, $D$ takes a machine $M$ and checks, using $H$, whether $M$ halts when run on the binary representation of itself; then it does the opposite of whatever $M$ does when run on itself. If $M$ halts on itself, $D$ will loop forever, and if $M$ loops forever, $D$ will halt.

Now suppose we run $D$ on itself. This creates a paradox: If $D$ halts when run on itself, $D$ will loop forever when run on itself. If $D$ loops forever when run on itself, then $D$ will halt when run on itself. Clearly, $D$ cannot exist! We arrived at this paradox by assuming that $H$ existed, since the existence of $H$ allowed us to build $D$. Since we can’t build $D$, $H$ must not exist! Thus, the language $A_{TM}$, often known as the Halting Problem, is undecidable.