Turing Machines
Applied Theory (CSCI 3500), Fall 2014

Prof. Harley Eades (heades@gru.edu).

Read chapter 3.1

So far we have seen several example models of computation. The first was finite automata, but we saw that they had limits because they do not have any sort of memory. We then studied context-free languages and PDAs which overcome some of the limitations of regular languages, but context-free languages contain limits themselves. PDAs have an unlimited amount of memory, but it can only be accessed in a last-in-first-out manner. In the next few lectures we will be studying yet another model of computation called Turing machines. These machines were invented by Alan Turing in the 1930's, and overcome all of the limitations we have seen. They are similar to PDAs with an unlimited amount of memory that can be arbitrarily accessed. In fact, Turing machines are theoretically as powerful as modern day computers.

Today computers are extremely powerful. One might even believe that any problem could be solvable by a computer, but this is not the case. Computers are theoretically as powerful – or computationally equivalent to – Turing machines which we will see have limits of their own. There are problems that Turing machines cannot solve which implies there are problems a computer cannot solve. One might expect these problems to be of only theoretical interest and cannot possibly be practically significant, but this is just not the case. Many simple practical problems are not solvable by a Turing machine and hence a computer.

One major practical problem that is unsolvable by computers lies within the scope of the authors own research area. Consider implementing some program, and as one developed this program one defined a mathematical specification as a list of lemmata. The program would be deemed correct if all of the lemmata in the specification are true. One might think that we could construct an algorithm that takes in the program, and the specification, and then outputs true if the program meets the specification, or false otherwise. Thus, automating the entire process. This problem is impossible for any computer to solve. We can prove this by showing that if we could solve this problem, them we could solve the much simpler sounding problem called the Halting problem. The latter problems asks for an algorithm that takes in as input an arbitrary program, and then outputs yes if the input program terminates, and outputs false otherwise. This problem is also impossible for a computer or a Turing machine to solve. Thus, the orignal problem we started with is also impossible. This is called reducing a problem into another. It is of the utmost importance for computer scientists to be able to characterize which problems are impossible so as to not waste time trying to solve problems that are not possible to solve.

1 The definition of a Turing machine

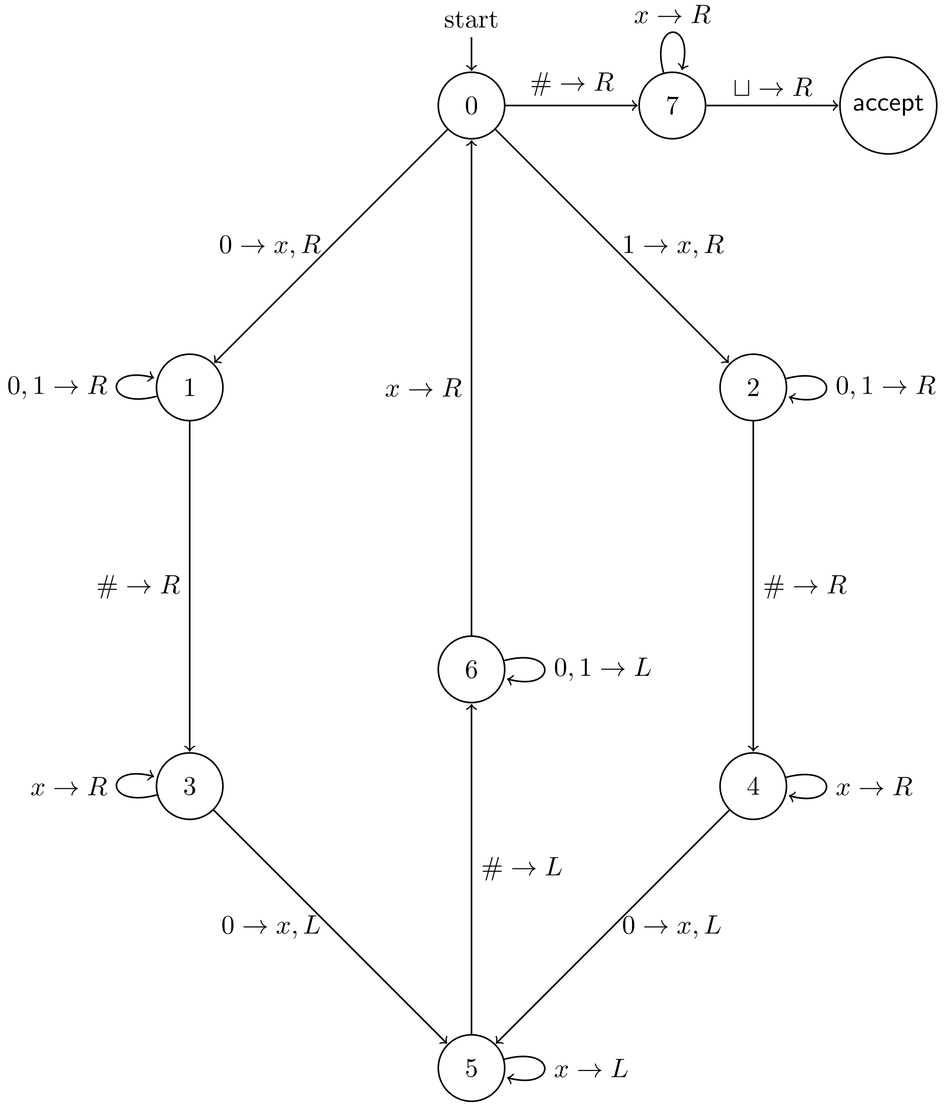
We now give the formal definition of a Turing machine.

Definition 1. *A **Turing machine** is a 7-tuple* $(Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$*:*

1. *Q is the finite set of states,*
2. *Σ is the finite input alphabet not containing the special **blank symbol** $\sqcup$,*
3. *Γ is the finite tape alphabet, where $\sqcup \in \Gamma$ and $\Sigma \subseteq \Gamma$,*
4. *$\delta : Q \times \Sigma \to Q \times \Gamma \times \{L, R\}$ is the transition function,*
5. *$q_0 \in Q$ is the start state,*
6. *$q_{accept} \in Q$ is the accept state, and*
7. *$q_{reject} \in Q$ is the reject state, where $q_{accept} \neq q_{reject}$.*

A Turing machine is run with respect to an infinite tape where the initial input to the machine is written on the left-most square, and the remainder of the tape is blank – each square has the blank symbol written on it.

Consider the language $L = \{w\#w \mid w \in \{0,1\}^*\}$. This language can be decided by the following Turing Machine:



The above example illustrates how the diagrammatic presentation of Turing machines looks. It is a straightforward extension of the diagrams for pushdown automata. Each transition has the form $a \rightarrow b, R$ or $a \rightarrow b, L$. The former reads "reading an a from the cell of the tape the head is placed, write a b and move right". The latter is similar. The transition label $a \rightarrow R$ is read "reading an a off of the tape, write an a and move Right." Sometimes we use the notion $a, b \rightarrow R$ which states "reading an a or a b, move right leaving the symbol on the tape the same." The situation is similar for the transitions labeled with $a \rightarrow L$ and $a, b \rightarrow L$.

Next we define how a Turing machine computes, but before we can do this we need to define the notion of a configuration.

Definition 2. *The **configuration** of a Turing machine $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$ is a snapshot of the current state the machine is in. That is, it is a snapshot of the current tape, state, and head position. For example, a configuration might looks like uaq_ibv, which states that the machine M is in state q_i, the head is reading b, and the tape contents is $uabv$. Configurations are exactly the same thing as the traces we have been using all semester. The major difference is the ability to move left and right instead of just right.*

The following are special forms of configurations:

- *The **start configuration** is q_0w for some $w \in \Gamma^*$.*
- *The **accepting configuration** is q_{accept}.*
- *The **rejecting configuration** is q_{reject}.*
- *We call a state in which the machine stops running a **halting configuration**.*

Finally, the formal definition of computation for Turing machines:

Definition 3. *A Turing machine $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$ computes as follows. Suppose $a, b, c \in \Gamma$, $u, v \in \Gamma^*$, and $q_i, q_j \in Q$.*

1. *If M is in configuration uaq_ibv and $\delta(q_i, b) = (q_j, c, L)$, then the M will enter the configuration uq_jacv.*
2. *If M is in configuration uaq_ibv and $\delta(q_i, b) = (q_j, c, R)$, then the M will enter the configuration $uacq_jv$.*
3. *If M is on the left-most square of the tape, and it is in configuration q_ibv and $\delta(q_i, b) = (q_j, c, L)$, then the M will enter the configuration q_jcv. This prevents the head of the machine from going off the end of the tape.*
4. *If M is on the left-most square of the tape, and it is in configuration q_ibv and $\delta(q_i, b) = (q_j, c, R)$, then the M will enter the configuration cq_jv.*

Just as we have seen before Turing machines recognize languages.

Definition 4. *The **language** of (or the **language recognized by**) a Turing machine is the set of words a Turing machine acceptes. We call a language **Turing-recognizable (or recursively enumerable)** if some Turing machine recognizes it.*

A Turing machine has the potential to loop forever. This makes determining if a Turing machine acceptes a word very hard, because we cannot tell if the machine is taking a really long time or looping. Now if by definition a Turing machine halts on all inputs, then we can always expect a response of accept or reject. These types of Turing machines form a special class:

Definition 5. *A Turing machine that halts on all input is called a **decider**. We call a language **Turing-decidable (decidable or recursive)** if it is the language of a decider.*

2 Examples

The first example shows that a Turing machine can recognize the following language:

$$L = \{w \mid w \in \{a, b\}^* \text{ and } w = a^n b^n\}$$

Whenever faced with the task of constructing a Turing machine it is best to first write down an informal algorithm describing how the machine will run, and then translate that algorithm to an actual machine.

The informal reasoning:

1. Given a word $w = a^n b^n$ write w on the left-most squares of the tape, and position the head on the left most square.
2. Move right skipping any square with an a on it until reaching the first square with a b on it. Move left.
3. Overwrite a with X and move right.
4. Move right skipping any X's until reaching the first b. Overwrite b with X and move left.
5. Move left until reaching the first a skipping all other symbols. If no a is found, then sweep right until either a b is reached or the first square with the blank symbol on it is reached. If the former occurs reject, otherwise accept. If an a is found, then repeat steps 3 through 5.

Here is a representation of what the above algorithm tries to describe. Suppose we write $|a|abb$ for the tape with $aabbb$ written on the left-most squares of the tape, and the head depicted as $|\_|$ where the symbol in the middle is what the head is reading. Then the above algorithm describes the following behavior:

$$\begin{array}{l}
|a|abbb \\
a|a|bbb \\
aa|b|bb \\
a|a|bbb \\
aX|b|bb \\
a|X|Xbb \\
|a|XXbb \\
X|X|Xbb \\
XX|X|bb \\
XXX|b|b \\
XX|X|Xb \\
X|X|XXb \\
|X|XXXb \\
X|X|XXb \\
XX|X|Xb \\
XXX|X|b \\
XXXX|b| \text{ (reject)}
\end{array}$$

Here is another example trace:

$|a|abb$
$a|a|bb$
$aa|b|b$
$a|a|bb$
$aX|b|b$
$a|X|Xb$
$|a|XXb$
$X|X|Xb$
$XX|X|b$
$XXX|b|$
$XX|X|X$
$X|X|XX$
$|X|XXX$
$X|X|XX$
$XX|X|X$
$XXX|X|$
$XXXX|\_|$ (accepted)

The diagram form of the Turing machine that recognizes the language $L = \{w\#w \mid w \in \{0,1\}^*\}$ is given above, but here is the informal reasoning. Suppose a word $w\#w \in L$ is written on the left most squares of the tape, and the head is positioned on the left most square. Then we have the following algorithm:

0. If the head is reading #, move right, and goto step 11.
1. Remember the current character the head is reading, mark it with an X, and move right.
2. Sweep right skipping all 0's, 1's, and X's until reading #.
3. Move right.
4. Sweep right skipping all X's.
5. If the head is reading the remembered character, then mark it, and move left. Otherwise, reject.
6. Sweep left skipping all X's until the head is reading #.
7. Move left skipping all 0's and 1's until the head is reading a X.
8. Move right.
9. If the head is not reading #, then repeat steps 0 - 8.
10. Sweep left skipping only X's.
11. If the head is reading a blank, then accept, otherwise reject.

3 What is an algorithm?

Introductory computer science courses introduce the notion of an algorithm as a basic set of instructions for the completion of a task. This task can be as simple as adding two numbers or as complex as the guidance system for a rocket launcher. However, what is the mathematical definition of an algorithm? How can we make it precise? This is a hard question to answer, but Alonzo Church and Alan Turing gave a precise definition in 1936 called the Church-Turing Thesis.

The basic statement of the Church-Turing thesis is as follows:

Intuitive notion of an algorithm	=	Turing machine algorithms	(Imperative Programming)
Intuitive notion of an algorithm	=	λ-calculus algorithms	(Functional Programming)

The rigorous definition of a Turing machines algorithm corresponds to the definition of the behavior of the Turing machine, and hence, the definition of the transition function. If the reader is interested in the functional programming and the λ-calculus, then they should think about taking the CSCI:3300 course on programming languages.

It is possible to define a Turing machine informally yielding a description of a Turing machine that is more amendable to algorithmic analysis. This description should be thought of like one thinks of pseudo code. As an example lets consider how to describe a Turing machine algorithm to compute the sum of two natural numbers. For the mathematically forgetful the natural numbers are elements of the set $\mathbb{N} = \{0, 1, 2, 3, \ldots\}$.

First, we must decide how to encode the input to our algorithm as a language. There are lots of encodings of the natural numbers we can choose from. Some encodings are easier than others. For example, the language $B_{\mathbb{N}} = \{w \mid w \in \{0,1\}^* \text{ is the binary equivalent to } n \in \mathbb{N}\}$ is an example encoding. Designing a Turing machine to add two binary numbers is doable, but a little complex. There is an easier type of encoding.

We will encode natural numbers as peano numbers. The set of peano numbers, denoted $\mathbb{P}$, are defined by the following rules:

$$\frac{}{0 \in \mathbb{P}}\ \textsc{Zero} \qquad \frac{n \in \mathbb{P}}{(s\, n) \in \mathbb{P}}\ \textsc{Suc}$$

We call the unary function s the successor function, and it should be thought of as taking in a natural number n and then returning $n + 1$.

A list of the first ten peano numbers:

0. 0
1. $s\,0$
2. $s\,s\,0$
3. $s\,s\,s\,0$
4. $s\,s\,s\,s\,0$
5. $s\,s\,s\,s\,s\,0$
6. $s\,s\,s\,s\,s\,s\,0$
7. $s\,s\,s\,s\,s\,s\,s\,0$
8. $s\,s\,s\,s\,s\,s\,s\,s\,0$
9. $s\,s\,s\,s\,s\,s\,s\,s\,s\,0$
10. $s\,s\,s\,s\,s\,s\,s\,s\,s\,s\,0$

Therefore, a natural number $n \in \mathbb{N} = \{0, 1, 2, 3, 4, \ldots\}$ can be defined as the peano number $\hat{n} = s^n 0$ where $s^n\, 0 = \underbrace{s \cdots s}_{n}\, 0$.

Peano numbers are a particular representation of the natural numbers, in fact, they look a lot like a datatype. Next we try and understand how we might define addition.

The addition operation on peano numbers is best described if we think in a recursive fashion. Suppose we want to add the numbers $\hat{3}$ and $\hat{4}$ to obtain $\hat{7}$. How might we do this? We can start by noticing that:

$$\hat{3} = s\,s\,s\,0 \quad \hat{4} = s\,s\,s\,s\,0$$

Now we want to obtain $\hat{7} = s\,s\,s\,s\,s\,s\,s\,0$, but notice that

$$s\,s\,s\,s\,s\,s\,s\,0 = s\,s\,s\,(s\,s\,s\,s\,0) = s\,s\,s\,(s^4\,0) = s^3\,(s^4\,0) = s^{3+4}\,0$$

How might we do this recursively? We can peal off a successor from the first number and add it to the second:

$$\begin{array}{ll} \hat{3} = s\,s\,s\,0 & \hat{4} = s\,s\,s\,s\,0 \\ \hat{2} = s\,s\,0 & \hat{5} = s\,s\,s\,s\,s\,0 \\ \hat{1} = s\,0 & \hat{6} = s\,s\,s\,s\,s\,s\,0 \\ \hat{0} = 0 & \hat{7} = s\,s\,s\,s\,s\,s\,s\,0 \end{array}$$

Then the final solution is on the right when we hit 0 on the left.

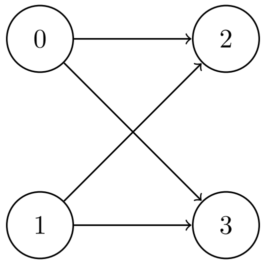
At this point we can take our informal analysis above, and describe a Turing machine informally. We do this next.

Definition 6. *The following defines a Turing machine algorithm for adding two peano numbers:*

```
M = "On input $\langle p_1\#p_2\rangle$, the encoding of the two peano numbers:
    0. Repeat the following until all characters of $p_1$ are marked:
        1. If $p_1 = 0$, then mark it and accept.
        2. For each successor of $p_1$ mark the successor and replace the $0$ of $p_2$ with $s\,0$.
    3. Reject.
```

The above algorithm uses "accept" as "return", and "reject" as "failure." If "accept" is reached, then the tape should look like $X\cdots X\#s^{p_1+p_2}\,0$. The final solution is on the right of the pound sign. The language this Turing machine algorithm decides is the singleton language $A_{p_1,p_2} = \{s^{p_1+p_2}\,0\}$ for some peano numbers p_1 and p_2.

It is possible to encode other types of data structures as languages as well. A directed graph (or just a graph) is a pair (N, E) of a set of nodes N and a binary relation $E \subseteq N \times N$. How might we encode graphs as a language? The words of the language should represent graphs. Consider the following graph:



We can represent this graph as the word $(0,1,2,3)((0,2),(0,3),(1,2),(1,3))$. Now we can define the language of graphs as being the language:

$$\mathsf{Graph} = \{(n_1,\cdots,n_i)((n_x,n_y),\cdots,(n_{x'},n_{y'})) \mid n_x,n_y,n_{x'},n_{y'} \in \{n_1,\ldots,n_i\} \text{ and each } n_j \in (\Sigma_N \cup \{(,)\})^*\}$$

where Σ_N is the alphabet for describing the nodes of the graph.

4 Decidability

One of the most important definitions given above is that of a decidable language. A language is decidable if and only if it is the language of a decider. Compose this definition with the definition of an algorithm (Church-Turing Thesis) and we now have that a language is decidable if and only if there exists a terminating algorithm that decides it.

The previous point is the pinnacle of what we have been studying in this course. It states that if a problem is solvable by modern day computational devices, then there must exist a terminating algorithm that solves the problem. Thus, we now have a criteria for determining when a problem is computationally solvable or not. We have yet to encountered undecidable problems (problems that cannot be solved by any algorithm), and we will be studying these below, but first we consider how to prove a language is decidable.

Consider the language:

$$\mathsf{DFA} = \{\langle M, w\rangle \mid M \text{ is a DFA that accepts the input word } w\}$$

The question: is this language decidable? To answer this question we need to define an algorithm that decides it. We use the informal representation used above when we solved the addition problem.

The proof of the decidability of DFA can be summarized as first taking in a word $\langle M, w\rangle$ as input, then simulating M on input w, and then accepting when M accepts and rejecting otherwise. The implementation details are similar to the implementation details we encountered in the first course project.

Speaking of the project we have already proven that the following languages are decidable:

$$\mathsf{NFA} = \{\langle M, w\rangle \mid M \text{ is a DFA that accepts the input word } w\}$$

$$\mathsf{REG} = \{\langle R, w\rangle \mid R \text{ is a regular expression that matches the input word } w\}$$

The proofs are the solutions to the first project. We have also proved the decidability of the following language:

$$\mathsf{CFG} = \{\langle G, w\rangle \mid G \text{ is a CFG that generates the input word } w\}$$

The proof is the solution to the extra credit project.

5 Undecidability

Consider the language:

$$\mathsf{A_{TM}} = \{\langle M, w\rangle \mid M \text{ is a Turing machine that accepts } w\}$$

Is the following Turing machine a decider for $\mathsf{A_{TM}}$?

```
$U$ = "On input $\langle M, w\rangle$, where $M$ is a TM and $w$ is a word over $M$'s alphabet:
    1. Simulate $M$ on input $w$.
    2. If $M$ ever enters its accept state, accept; if $M$ ever enters its reject state, reject."
```

The answer is no, because M could run forever, and hence U will run forever. How do we prove this? The previous construction shows that $\mathsf{A_{TM}}$ is Turing recognizable; U is called the universal Turing Machine, because it can simulate any other Turing Machine.

To prove that $\mathsf{A_{TM}}$ is undecidable we first must understand a proof technique called the **Diagonalization Argument** which was first proposed by Gregory Cantor. However, this technique requires some elementary facts about functions and sets.

Definition 7. *A function $f : A \to B$ is* ***injective*** *(or one-to-one) iff for all $a_1, a_2 \in A$, $f(a_1) = f(a_2)$ implies $a_1 = a_2$.*

Definition 8. *A function $f : A \to B$ is* ***surjective*** *(or onto) iff for all $b \in B$, there exists at least one $a \in A$ such that $f(a) = b$.*

Definition 9. *A function $f : A \to B$ is called a* ***bijection*** *iff it is injective and surjective.*

Now using the previous notions we arrive at the following definitions:

Definition 10. *Two potentially infinite sets A and B are equal in size iff there exists a bijection $f : A \to B$.*

Definition 11. *A set A is* ***countable*** *iff it is finite, or has the same size as $\mathbb{N}$.*

Example 12. *Suppose $\mathcal{O} = \{1, 3, 5, 7, 9, 11, \ldots\}$ is the set of all odd natural numbers. The set $\mathcal{O}$ is countable. To show this we must exhibit a bijection $f : \mathbb{N} \to \mathcal{O}$. Set $f(n) = 2n + 1$. Clearly, f is a bijection whose inverse is $f^{-1}(y) = (y - 1)/2$. Therefore, there are the same number of odd natural numbers as there are natural numbers.*

Example 13. *Consider the set $\mathcal{Q} = \{\frac{m}{n} \mid m, n \in \mathbb{N}\}$ of all rational numbers. Now construct the infinite matrix consisting of rational numbers where the ith row of the matrix contains all the rational numbers with i for their numerator, and the jth column to be all of the rational numbers with j as their denominator. This matrix has the following form:*

$$\begin{array}{cccccc}
\frac{1}{1} & \frac{1}{2} & \frac{1}{3} & \frac{1}{4} & \frac{1}{5} & \cdots \\
\frac{2}{1} & \frac{2}{2} & \frac{2}{3} & \frac{2}{4} & \frac{2}{5} & \cdots \\
\frac{3}{1} & \frac{3}{2} & \frac{3}{3} & \frac{3}{4} & \frac{3}{5} & \cdots \\
\frac{4}{1} & \frac{4}{2} & \frac{4}{3} & \frac{4}{4} & \frac{4}{5} & \cdots \\
\frac{5}{1} & \frac{5}{2} & \frac{5}{3} & \frac{5}{4} & \frac{5}{5} & \cdots \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots
\end{array}$$

Using the previous matrix we now construct the infinite list consisting of non-repeated rational numbers. The list is constructed by moving along the upper-left most diagonals – the colored diagonals – of the previous matrix. The first element of the list is $\frac{1}{1}$ which is the upper-left most diagonal of the matrix. Next we move to the blue diagonal and set the second element of the list to $\frac{2}{1}$ and then $\frac{1}{2}$. The forth element is $\frac{3}{1}$, but we must skip the element $\frac{2}{2}$, because it is already accounted for in the list, namely by the element $\frac{1}{1} = 1 = \frac{2}{2}$, and so the fifth element is $\frac{1}{3}$. The next elements are $\frac{4}{1}$, $\frac{3}{2}$, $\frac{2}{3}$, $\frac{1}{4}$, $\frac{5}{1}$, but we skip all of the other elements of the orange diagonal except $\frac{1}{5}$. Now if we proceed in this fashion we will end up with the infinite list:

$$\frac{1}{1} \quad \frac{2}{1} \quad \frac{1}{2} \quad \frac{3}{1} \quad \frac{2}{2} \quad \frac{1}{3} \quad \frac{4}{1} \quad \frac{3}{2} \quad \frac{2}{3} \quad \frac{1}{4} \quad \frac{5}{1} \quad \cdots$$

Keep in mind that we skipped repeats and so the previous list consists of all the unique rational numbers. Using the positions of the elements of the list we can construct a bijection between the natural numbers and the rational numbers:

$$\begin{array}{cccccccccccc}
\frac{1}{1} & \frac{2}{1} & \frac{1}{2} & \frac{3}{1} & \frac{2}{2} & \frac{1}{3} & \frac{4}{1} & \frac{3}{2} & \frac{2}{3} & \frac{1}{4} & \frac{5}{1} & \cdots \\
\updownarrow & \updownarrow & \updownarrow & \updownarrow & \updownarrow & \updownarrow & \updownarrow & \updownarrow & \updownarrow & \updownarrow & \updownarrow & \cdots \\
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & \cdots
\end{array}$$

We have now shown that there are exactly the same number of rational numbers as there are natural numbers. That is, $|\mathcal{Q}| = |\mathcal{N}|$. This may seem quite surprising.

The previous examples use a bijection to establish that two sets are of the same size, but we can relax this using the following result.

Theorem 14. *A set A is* ***countable*** *if and only if there is an injection between A and $\mathbb{N}$.*

An injection from a set A and the natural numbers establishes that every element of A is mapped to a unique natural number. Thus, there is at most a countably infinite number of elements in A. The relaxation to injections instead of bijections makes proving a set countable easier. We use this in the following proofs.

Theorem 15. *Suppose A and B are countable sets. Then $A \times B$ is countable.*

Proof. If A and B are finite, then $A \times B$ is finite, and hence, countable. So suppose A and B are countably infinite. Then there exist bijections $b_1 : A \to \mathbb{N}$ and $b_2 : B \to \mathbb{N}$. We know that given two functions we can produce the function $b_1 \times b_2 : A \times B \to \mathbb{N} \times \mathbb{N}$, furthermore, it is a bijection. Thus, if we can produce an injection $b_3 : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ then we can produce our desired injection by composing $b_1 \times b_2$ with b_3.

We now prove that b_3 is an injection. We define $b_3(n, m) = 2^n 3^m$. We know by the fundamental theorem of arithmetic that if we can write a number as a prime decomposition then we know that it is unique, this implies that b_3 is injective.

Finally, we know that the composition of two injective functions is again injective, and thus, $b_3 \circ (b_1 \times b_2) : A \times B \to \mathbb{N}$ is injective. Therefore, $A \times B$ is countable. $\square$

Theorem 16. *Suppose A and B are countable sets. Then $A \cup B$ is countable.*

At this point it is natural to ask, is there a set that is larger than the natural numbers? That is, is there an **uncountable** set? Consider the set of real numbers denoted $\mathbb{R}$. It turns out that this set is indeed larger than the natural numbers. To prove this we must show that there exists no bijection between the natural numbers and $\mathbb{R}$.

The proof of the uncountability of $\mathbb{R}$ is a little more complicated than the previous proofs. It requires a new mathematical tool called the diagonalization method which was first proposed by Gregory Cantor. He introduced the method precisely to prove that there are more real numbers than natural numbers. It is a very powerful method.

Theorem 17. *The set $\mathbb{R}$ is uncountable.*

Proof. We must show that there does not exist any bijection between the sets $\mathbb{N}$ and $\mathbb{R}$. To do this we assume there is a bijection and reach a contradiction. So suppose $b : \mathbb{N} \to \mathbb{R}$ is a bijection. To reach the contradiction we use Cantor's diagonalization method. We will contradict the fact that b is a bijection by finding an $r \in \mathbb{R}$ such that there is no $n \in \mathbb{N}$ where $b(n) = r$, and thus, b is not a bijection.

The beauty of this method is that it does not merely prove the existence of r, but actually constructs it. We have assumed that b is a bijection, and thus, we know we can build the following table of inputs and outputs:

n	$b(n)$
1	$a_1.a_2a_3a_4a_5\cdots$
2	$b_1.b_2b_3b_4b_5\cdots$
3	$c_1.c_2c_3c_4c_5\cdots$
4	$d_1.d_2d_3d_4d_5\cdots$
5	$e_1.e_2e_3e_4e_5\cdots$
$\vdots$	$\vdots$

Now we use the previous table to construct a real number r such that $0 < r < 1$ and there is no natural number n where $b(n) = r$. We will choose each decimal of r by examining the table given above. Set the first decimal of r to be any number different from the first decimal number of $f(1)$ which is a_2. So the first decimal number of r is any number different from a_2, and thus, $r \neq f(1)$. Set the second decimal number of r to be any number different from the second decimal of $f(2)$ which is b_3. Thus, the second decimal of r is not b_3, and thus, $r \neq f(2)$. Now set the third decimal number of r to be any number different from the third decimal of $f(3)$ which is c_4. Thus, the third decimal of r is not c_4, and thus, $r \neq f(3)$. In general, set the nth decimal of r to be any number different from the nth decimal of $f(n)$, and thus, $r \neq f(n)$ for any n. Clearly, r constructed in this way is a real number between 0 and 1, and there does not exist a natural number n where $b(n) = r$, because we constructed it to be different from every real number that could have been mapped. Thus, b cannot be a bijection; a contradiction. Therefore, there are more real numbers than natural numbers. □

We can use the previous theorem to prove that there are more undecidable languages than decidable ones, and there are some languages that are not even Turing-recognizable. First, we prove that latter.

Theorem 18. *There are languages that are not Turing-recognizable.*

Proof. To prove this we will first show that the set of all Turing machines – and hence Turing-recognizable languages – is countable. Then we will form a bijection between the set of all languages and the set of real numbers, and hence, conclude that there are uncountably many languages. Thus, we will be able to conclude by the previous theorem that there is no bijection between the set of Turing-recognizable languages and the set of all languages.

We now show that there are countably many Turing machines, and hence, countably many Turing-recognizable languages. First, notice that given an alphabet Σ the language Σ^* of all words over Σ is countable. Form the following disjoint sets: $\Sigma^n = \{w \mid w \in \Sigma^* \text{ and } |w| = n\}$. Then we know that $\bigcup_{n\in\mathbb{N}} \Sigma^n = \Sigma^*$. Now we know that each of Σ^n is finite, and thus, by Theorem 16 we know that $\bigcup_{n\in\mathbb{N}} \Sigma^n$ is countable, and thus, Σ^* is as well. Using this fact we can easily see that the set of Turing machines is countable, because we can encode a Turing machine as a string over some alphabet Σ, and thus, the set of all valid strings encoding Turing machines is a strict subset of Σ^*, and therefore, is countable.

Now we show that there are uncountably many languages, and thus, there exists languages that are not Turing-recognizable. Consider the set of all infinite binary sequences $\mathcal{B}$. Using the diagonalization method we can show that $\mathcal{B}$ is uncountable. In fact, the proof is very similar to the proof that the set of reals is uncountable.

Let $\mathcal{L}$ be the set of all languages over the alphabet Σ. Suppose $\Sigma^* = \{s_1, s_2, s_3, \ldots\}$. We construct a bijection $b : \mathcal{L} \to \mathcal{B}$ by constructing a unique sequence of binary digits for each language. Suppose $A \in \mathcal{L}$. Then we construct the binary sequence $r \in \mathcal{B}$ as follows: the ith bit of r is 1 if $s_i \in A$ and 0 otherwise. This is known as the **characteristic sequence** for the language A. Thus, we define $b(A) = r$. This is clearly a bijection, because the sequence is unique. Thus, the set of all languages is uncountable! □

The previous theorem shows that there are vastly more problems (languages) than programs (Turing Machines). The next result shows that the language $\mathsf{A_{TM}}$ is undecidable which gives us our first example of a problem that cannot be solved by a program in general. That is, there is no decider that decides it. However, we have already shown that it is indeed Turing recognizable. This language is also interesting, because it can be used to construct a language that is not Turing recognizable which is an example of a computationally unsolvable problem.

Theorem 19. *The language* $\mathsf{A_{TM}}$ *is undecidable.*

Proof. We assume $\mathsf{A_{TM}}$ is decidable and derive a contradiction. This implies that there must exist a decider, H, that decides $\mathsf{A_{TM}}$. Now we construct a second TM that uses H as a subroutine:

```
$D$ = “On input $\langle M \rangle$, where $M$ is a TM:
        1. Simulate $H$ on input $\langle M, \langle M \rangle \rangle$.
        2. If $H$ ever enters its accept state, reject; if $M$ ever enters its reject state, reject.”
```

Now run D on itself, $D(\langle D \rangle)$, but if D accepts then D rejects, and if D rejects, then D accepts. Clearly, we have reached a contradiction, because no matter what D does $H(\langle D \rangle)$ does the opposite. Therefore, neither D nor H can exist. □

6 A Turing-Unrecognizable Language

In this section we show that there exists a problem such that no solution for the problem exists that can be programmed on any computational device. We first need a definition.

Definition 20. *A language is* ***co-Turing-recognizable*** *if and only if it is the complement of a Turing-recognizable language.*

We now have the following result.

Theorem 21. *A language is decidable if and only if it is Turing-recognizable and co-Turing-recognizable.*

Finally, using the previous result and the fact that we know that A_{TM} is undecidable (Theorem 19) we can prove the following:

Theorem 22. *The language* $\overline{\mathsf{A}_{\mathsf{TM}}}$ *is Turing-unrecognizable.*

Proof. We have already shown that A_{TM} is Turing-recognizable, and thus, if $\overline{\mathsf{A}_{\mathsf{TM}}}$ were co-Turing-recognizable then A_{TM} would be decidable, but this would contradict Theorem 19. Therefore, $\overline{\mathsf{A}_{\mathsf{TM}}}$ is Turing-unrecognizable. □