Computation

P. Danziger

1 Finite State Automata (12.2)

Definition 1 *A* Finite State Automata *(FSA) is a 5-tuple* $(Q, \Sigma, \delta, q_0, F)$ *where:*

- Q is a finite set, called the set of states. The elements of Q are called *states*
- Σ is a finite set, called the *alphabet* of the FSA.
- $\delta : Q \times \Sigma \to Q$ is a function, called the transition or state function. $\delta(q, a) = q',\ q, q' \in Q$ and $a \in \Sigma$.
- $q_0 \in Q$ is the *start state*.
- $F \subseteq Q$ is the set of *accept sates*.

The machine begins in the start state q_0 and is provided an *input string*, which it reads sequentially, one character at a time. If the machine is in state $q \in Q$ and reads character $a \in \Sigma$ the machine moves to state $\delta(q, a)$ and continues to read from there.
If after reading the final character the machine is in an accept state (one of the states in F) then the machine *accepts* the input string, otherwise we say that the machine *rejects* the input string.
FSAs thus parse strings either accepting them as "good" or rejecting them as "bad".

Definition 2 *Given an FSA* M*, the set of strings accepted by* M *is called the* language accepted by M *or the language* recognized *by* M *and is denoted* $L(M)$.
We say that two FSA are equivalent *if they accept the same language. ie.* $M_1 \equiv M_2$ *if and only if* $L(M_1) = L(M_2)$.

Given an FSA $M = (Q, \Sigma, \delta, q_0, F)$ it is useful to define the *eventual state function* $\delta^* : Q \times \Sigma^* \to Q$, $\delta^*(q, x)$ gives the state the machine M would be in if it started in state q and processed the string x. Thus, given a machine M and $w \in \Sigma^*$, $w \in L(M)$ if and only if $\delta^*(q_0, w) \in F$.
The main task in defining an FSA is to define the state function δ. There are essentially two ways to do this

State Table Since δ is a finite function we can write out a table of all possible inputs and give the resultant output.

State Diagram In a state diagram each state is represented by a node in a digraph. There is an arc between the two nodes q and q' labelled a if $\delta(q, a) = q'$, ie. the machine in state q will move to state q' if it reads an a.

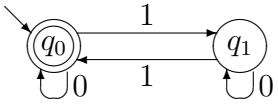
Example 3

$$\Sigma = \{0, 1\},\ Q = \{q_0, q_1\},\ F = \{q_0\}.$$

State Table

δ	0	1
q_0	q_0	q_1
q_1	q_1	q_0

State Diagram



Consider the action of this machine on the string 011:

$$\begin{array}{cccc} 011 & 011 & 011 & 011 \\ \uparrow & \uparrow & \uparrow & \uparrow \\ q_0 & q_0 & q_1 & q_0 \end{array}$$

$$\delta(q_0, 0) = q_0,\ \delta(q_0, 1) = q_1,\ \delta(q_1, 1) = q_0,$$

or more succinctly $\delta^*(q_0, 011) = q_0$. Since the Machine ends in an accepting state ($q_0 \in F$) the string is accepted, i.e. $011 \in L(M)$
As long as this machine reads 0's it atays in an accepting state (q_0). Whenever it reads a 1 it moves to a non accepting state (q_1), and stays there until it reads another 1. $L(M) = (0^*10^*1)^*0^* =$ strings with an even number of 1's.

Definition 4 *A language which is accepted by an FSA M is called regular. That is a language L is regular if and only if there exists a machine M such that $L = L(M)$.*

Theorem 5 (Kleene's Theorem) *A language is regular if and only if there is a regular expression for it.*

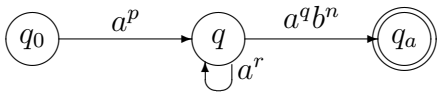
That is For every language defined by a regular expression there is an FSA which accepts it and every language accepted by some FSA can be expressed by a regular expression.
This means that there is a bijection between the set of FSA and the set of regular expressions, we can blur the distinction between the language and the machine that recognizes it.

2 Non Regular Languages

A natural question is weather there are indeed languages which are not accepted by any FSA. There are indeed such languages and we now exhibit one.

Theorem 6 *The language $L = \{a^n b^n \mid n \in \mathbb{Z}\}$ is not accepted by any FSA.*

Proof: (By contradiction)Suppose M is an FSA which accepts L with m states and L has the minimum number of states of all FSA's that recognize L. Choose $n \in \mathbb{Z}$ with $n > m$, and consider the action of M on the string $w = a^n b^n \in L$. Since $w \in L$, M must end in an accept state q_a say, i.e. $\delta^*(q_0, w) = q_a \in F$. Now since $n > m$, by the pigeonhole principle there must be a state q which is repeated as M processes the substring a^n, i.e. there exists $r \in \mathbb{Z}^+$ such that $\delta^*(q, a^r) = q$.



Where $p + r + q = n$. (The arcs on the diagram above represent paths in the actual machine. For example the path labeled a^p from q_0 to q indicates that if the machine is in state q_0 and reads a^p then it will go to state q, i.e. $\delta^*(q_0, a^r) = q$.)
But now consider the action of M on the string $a^{n+r}b^n = a^{p+2r+q}b^n$. M cannot distinguish this string from the original. It reads the first p a's, which takes it from state q_0 to q ($\delta^*(q_0, a^p) = q$). It then reads r more a's twice, ($\delta^*(q, a^r) = \delta^*(q, a^{2r}) = q$). Finally it reads the rest of the string ($a^r b^n$) to end up in state q_a ($\delta^*(q, a^r b^n) = q_a$). Thus $\delta(q_0, a^{n+r}b^n) = q_a \in F$. So M accepts $a^{n+r}b^n$, but $a^{n+r}b^n \notin L$ (since $r > 0$ $n + r \neq n$).
In fact this idea can be generalised into a theorem called the pumping lemma.

Theorem 7 (The Pumping Lemma) *Let L be any regular language, then there is a constant p, called the pumping constant for L, such that for any string $w \in L$ with $|w| \geq p$, w can be split into 3 substrings, x, y and z such that $w = xyz$, $|xy| \leq p$, $|y| > 0$ and for every $n \in \mathbb{N}$, $w_n = xy^n z \in L$.*

The Pumping Lemma is very useful for showing that a language is not regular. We do this by contradiction:
We start by assuming that L is a regular language.
Find a string $w \in L$ such that $|w| \geq p$, ideally the first p characters of w are the same (a say).
Since $|w| \geq p$ the pumping lemma applies to w, so $w = xyz$ as above. Further, since $|xy| \leq p$ and the first p characters of w are a we know that xy is a string of a's. So $x = a^s$, where $p \geq 0$ and $y = a^r$ for some $r > 0$.
Now, find $n \in \mathbb{N}$ such that $w_n = a^{s+nr}z \notin L$.
This contradicts the Pumping Lemma, so the assumption is false and L is not regular.

Example 8

1. $L = \{0^n 1^n \mid n \in \mathbb{N}\}$

   Assume that L is regular, so the Pumping Lemma applies to L.
   Let p be the pumping constant for L.
   Consider $w = 0^p 1^p$, $|w| \geq p$, so the Pumping Lemma applies to w.
   Let $w = xyz$, as in the pumping Lemma.
   Thus $|xy| \leq p$ and $|y| > 0$, which means that $y = 0^r$ for some $r > 0$.
   Now consider $w_2 = xyyz = 0^{p+r}1^p$, since $r > 0$, $p + r \neq n$ and so $w_2 \notin L$.
   This contradicts the Pumping Lemma and so L is not regular.

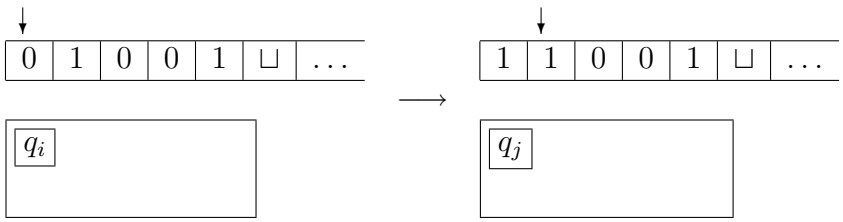
2. Palindromes over $\{0.1\}$, $L_{\text{pal}} = \{x \mid x \in \{0, 1\} \text{ and } x = x^R\}$, where x^R is the reversal of x.

   Assume that L is regular, so the Pumping Lemma applies to L.
   Let p be the pumping constant for L.
   Consider $w = 0^p 1 0^p$, $|w| \geq p$, so the Pumping Lemma applies to w.
   Let $w = xyz$, as in the pumping Lemma.
   Thus $|xy| \leq p$ and $|y| > 0$, which means that $y = 0^r$ for some $r > 0$.
   Now consider $w_2 = xyyz = 0^{p+r}10^p$, since $r > 0$, $p + r \neq n$ and so $w_2 \notin L$.
   This contradicts the Pumping Lemma and so L is not regular.

3 Turing Machines

We have seen how there is a correspondence between types of languages and the types of machines that accept them. We now consider the most powerful model of computation available Turing Machines and see what they imply about computation.
A Turing machine consists of a Finite State Control, which is an FSA, and an infinitely long read write 'tape'. This tape is divided into cells, at each step the read/write head is positioned over a particular cell.
The tape alphabet of a Turing Machine has a special symbol, often denoted $\sqcup$, or $\flat$, which indicates that a cell on the tape is blank.
A Turing Machine has two special states q_{accept} and q_{reject}.
If the machine ever enters the "accept" state, q_{accept}, it signals acceptance and halts processing.
If the machine ever enters the "reject" state, q_{reject}, it signals reject and halts processing.
Note that the only way that a Turing machine will halt is by entering one of these states, so it is possible that a Turing machine will continue processing forever and never halt.
Initially the tape contains the input string, and the tape read/write head is positioned over the leftmost symbol of the input string. At each step the Turing Machine performs the following actions:

1. Reads the current symbol from the tape.
2. Writes a symbol to the tape at the current position.
3. Moves to a new state in the Finite State Control.
4. Moves the read/write head either left or right one cell.



Note Unlike FSAs there is no requirement that a Turing machine read the input string sequentially, even if it does it may continue computing indefinitely (until it enters either q_{accept} or q_{reject}).

Definition 9 (Deterministic Turing Machine) *A Turing Machine, M, is a 7-tuple* $(Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$*, where* Q, Σ, Γ *are finite sets and:*

1. Q *is the set of states of M.*
2. Σ *is the input alphabet of M. The blank symbol* $\sqcup \notin \Sigma$.
3. Γ *is the tape alphabet of M.* $\sqcup \in \Gamma$ *and* $\Sigma \subseteq \Gamma$.
4. $\delta : Q \times \Gamma \longrightarrow Q \times \Gamma \times \{L, R\}$ *is the transition function of M.*
5. $q_0 \in Q$ *is the start state of M.*

6. $q_{accept} \in Q$ *is the accept state of* M.
7. $q_{reject} \in Q$ *is the reject state of* M.

Initially the tape contains the input string, and the tape read/write head is positioned over the leftmost symbol of the input string. The rest of the tape is filled with the blank symbol ($\sqcup$). The first blank thus marks the end of the initial input string.
The transition function then tells the machine how to proceed:

$$\delta(q_i, a) = (q_j, b, L)$$

Means: If we are in state q_i and we read an $a \in \Gamma$ at the current tape position, then move the finite state control to state q_j, write $b \in \Gamma$ and move the tape head left. (R means move the tape head right.)

Example 10

1. $M_1 = (\{q_0, q_{\text{accept}}, q_{\text{reject}}\}, \{0,1\}, \{0,1,\sqcup\}, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$

   $\delta(q_0, 0) = (q_{\text{accept}}, 0, R)$
   $\delta(q_0, 1) = (q_{\text{reject}}, 1, R)$
   $\delta(q_0, \sqcup) = (q_{\text{reject}}, 1, R)$.

   This machine goes to the accept state if 0 is the first character of the input string, otherwise it goes to reject.

2. $M_2 = (\{q_0, q_1, q_{\text{accept}}, q_{\text{reject}}\}, \{0,1\}, \{0,1,\#,\sqcup\}, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$

   $\delta(q_0, 0) = (q_{\text{accept}}, 0, R)$
   $\delta(q_0, 1) = (q_1, 1, R)$
   $\delta(q_0, \#) = (q_{\text{reject}}, 1, R)$
   $\delta(q_0, \sqcup) = (q_{\text{reject}}, 1, R)$.

   $\delta(q_1, 0) = (q_1, \#, R)$
   $\delta(q_1, 1) = (q_1, \#, R)$
   $\delta(q_1, \#) = (q_1, \#, R)$
   $\delta(q_1, \sqcup) = (q_1, \#, R)$.

   (This can be summarized as $\forall a \in \Gamma, \delta(q_1, a) = (q_1, \#, R)$.)

   This machine goes to the accept state if 0 is the first character of the input string, it goes to the reject state if the input string is empty. If the string starts with a 1, it tries to fill up the infinite tape with #'s, which takes forever.

We can represent the action of a Turing machine on a given input by writing out the current tape contents, with the state to the left of the current read/write head position.
Thus if we consider the action of M_2 on the string 10001:

$$\begin{array}{llll} q_0 10001\sqcup & \longrightarrow & \#q_1 0001\sqcup & \longrightarrow \\ \#\#q_1 001\sqcup & \longrightarrow & \#\#\#q_1 01\sqcup & \longrightarrow \\ \#\#\#\#q_1 1\sqcup & \longrightarrow & \#\#\#\#\#q_1 1\sqcup & \longrightarrow \\ \#\#\#\#\#\#q_1\sqcup & \longrightarrow & \#\#\#\#\#\#\#q_1\sqcup & \ldots \end{array}$$

Definition 11

1. *A string $w \in \Sigma^*$ is accepted by a Turing machine M if the machine enters the q_{accept} state while processing w.*
2. *The language $L(M)$ accepted by a Turing machine M is the set of all strings accepted by M.*
3. *A string is rejected by a Turing machine M if either M enters the q_{reject} state while processing w, or if M never halts in the processing of w.*
4. *A language L is called Turing recognizable or Recursively Enumerable if there exists some Turing machine, M, such that $L = L(M)$.*
5. *A language L is called Turing decidable orRecursive if there exists some Turing machine, M, such that $L = L(M)$, and M is guaranteed to halt on any input.*

Note The difference between Recognizable and Decidable is that in the latter case the machine is guaranteed to halt.
The problem of showing that there are languages which are Recognizable but not Decidable, is essentially the halting problem.

Example 12

1. Consider the machines M_1 and M_2 above.

   Clearly $L(M_1) = L(M_2) = 0(0 \vee 1)^* = L =$ any string beginning with 0.

   M_1 decides L, since it is guaranteed to stop.
   On the other hand M_2 only recognizes L, since it does not halt on any string beginning with a 1.

2. Design a Turing machine to decide the language $L = 0^n 1^n$.

   We introduce a tape character '#', to denote that the original symbol in a cell has been used. By saying it is 'crossed off' we mean replaced by '#'.

   $Q = \{q_0, q_1, q_2, q_3, q_4, q_{\text{accept}}, q_{\text{reject}}\}, \Sigma = \{0, 1\}, \Gamma = \{0, 1, \#, \sqcup\}$

   Algorithm:
   (a) Cross off the leftmost 0.
   (b) Scan right to end of input, cross off the rightmost 1.
   If there is no such 1 reject.
   (c) Scan left until we reach the leftmost surviving 0.
   If there is no such 0 scan right, if a 1 is encountered before reaching the end of the string reject, otherwise accept.

$\delta(q_0, \sqcup) = (q_{\text{accept}}, \sqcup, R)$ – Accept the empty string
$\delta(q_0, 1) = (q_{\text{reject}}, 1, R)$ – Reject any string which starts with 1
$\delta(q_0, 0) = (q_1, \#, R)$ – Cross off leftmost 0

$$\left.\begin{array}{rcl} \delta(q_1, 0) & = & (q_1, 0, R) \\ \delta(q_1, 1) & = & (q_1, 1, R) \\ \delta(q_1, \#) & = & (q_2, \#, L) \\ \delta(q_1, \sqcup) & = & (q_2, \sqcup, L) \end{array}\right\} \text{Scan right to rightmost 'live' cell of input.}$$

$\delta(q_2, \#) = (q_2, \#, L)$ – Scan left over crossed off symbols
$\delta(q_2, 0) = (q_{\text{reject}}, 0, L)$ – If rightmost live char. not 1 reject
$\delta(q_2, 1) = (q_3, \#, L)$ – Cross off rightmost 1.

$\delta(q_3, 1) = (q_3, 1, L)$ – Scan left over the 1's
$\delta(q_3, 0) = (q_3, 0, L)$ – Scan left over the 0's
$\delta(q_3, \#) = (q_4, \#, R)$ – # marks left end

$\delta(q_4, 0) = (q_1, \#, R)$ – cross off leftmost 0 and start again.
$\delta(q_4, \#) = (q_{\text{accept}}, \#, R)$ – all done accept.
$\delta(q_4, 1) = (q_{\text{reject}}, 1, R)$ – No more 0's, but still got 1's.

The following should never be encountered, they are all set to go to q_{reject}.

$\delta(q_0, \#), \delta(q_2, \sqcup), \delta(q_3, \sqcup), \delta(q_4, \sqcup)$.

We now consider the action of this machine on the string 0011.

$$\begin{array}{llllllll} q_00011\sqcup & \longrightarrow & \#q_1011\sqcup & \longrightarrow & \#0q_111\sqcup & \longrightarrow & \#01q_11\sqcup & \longrightarrow \\ \#011q_1\sqcup & \longrightarrow & \#01q_21\sqcup & \longrightarrow & \#0q_31\#\sqcup & \longrightarrow & \#q_301\#\sqcup & \longrightarrow \\ q_3\#01\#\sqcup & \longrightarrow & \#q_401\#\sqcup & \longrightarrow & \#\#q_11\#\sqcup & \longrightarrow & \#\#1q_1\#\sqcup & \longrightarrow \\ \#\#q_21\#\sqcup & \longrightarrow & \#q_3\#\#\#\sqcup & \longrightarrow & \#\#q_4\#\#\sqcup & \longrightarrow & \text{Accept} & \end{array}$$

We can see from the above example that it is tedious to write out the transition function in full. Turing machines are very powerful, in fact the Church Turing thesis holds that they can implement any algorithm.

Often it is sufficient to write out an algorithm which describes how the Turing machine will operate. Of course if we are asked for a formal description, we **must** provide the transition function explicitly.

When writing out algorithms a common phrase is:
Scan right (or left) performing action until x is reached.

For our next example we describe a Turing machine which decides a language which we know to be context sensitive, but not context free.

3. Design a Turing machine which decides $L = \{x \in \{0\}^* \mid x = 0^{2^n}, n \in \mathbf{N}\}$.

   $\Sigma = \{0\}, \Gamma = \{0, \#, \sqcup\}$.

   Scan right along the tape crossing off every other 0, this halves the number of 0's.
   If there is only one 0, accept.
   If the number of 0's is odd reject. (Note the parity of 0's can be recorded by state.)

Scan back to the left hand end of the string.
Repeat.

Consider the action of this algorithm on the strings 00000000 (0^8), and 0000000 (0^7):

initially	00000000		
After first pass	# 0 # 0 # 0 # 0	initially	0000000
After second pass	# # # 0 # # # 0	After first pass	# 0 # 0 # 0 #
After third pass	# # # # # # # 0	Reject	
Accept			

3.1 Variations

There are several standard variations of the definition of Turing Machines which we will now investigate.

3.1.1 Multi-tape Turing Machines

A Multi-tape Turing Machine is a Turing machine which has more than one tape.
If the machine has k tapes, it is called a k–tape Turing Machine.
The only change is in the transition function, we read k inputs from the k tapes, and write k outputs, in addition each of the k read/write heads moves either Left or Right.

$$\delta \; : \; Q \times \Gamma^k \longrightarrow Q \times \Gamma^k \times \{L, R\}^k$$

Theorem If a language is recognized by some k tape Turing machine M, then it is recognized by some 1 tape Turing machine M'.

3.1.2 Computation of Integer Functions

In this variation $\Sigma = \{0\}$, the initial input is of the form 0^n, for some n. When the machine halts the tape holds the computed *output*, 0^m, for some m. Such a machine is called a Computational Turing machine.
Thus for each n there is a corresponding m, we may interpret this as the result of some function: $m = f(n)$.
We say that the function f is computable.
A further variation allows for more than one input variable. In this case $\Sigma = \{0, \#\}$ and input strings are of the form $0^n\#0^k$. The output, 0^m is then the result of $m = f(n, k)$
In this case the numbers n, k and m are being represented in unary notation, which is standard. However, it is possible to use $\Sigma = \{0, 1, \#\}$, and to represent the numbers n, k and m in binary.

Example 13 *Find a computational Turing machine which computes* $f(n, m) = n + m + 1$

The input is of the form $0^n\#0^m$, we merely erase the 1 and check that the input has the correct form.
$\delta(q_0, 0) = (q_0, 0, R)$ – Scan right for #
$\delta(q_0, \#) = (q_1, 0, R)$ – Found it, change it to a 0
$\delta(q_1, 0) = (q_1, 0, R)$ – Scan right for end of string

$\delta(q_1, \sqcup) = (q_{\text{accept}}, \sqcup, R)$ – Found it, accept
Every other transition goes to q_{reject}.
It is worth noting that for all the many variations of Turing machines that have been investigated, **none** are more powerful (i.e. able to recognize more languages) than a standard Turing machine.

3.2 The Church Turing Thesis

Turing machines are extremely powerful in their computational abilities. They can recognize addition, subtraction, multiplication and division, exponentiation, (integer) logarithms and much more. Any operation which a PC can do can be done on a Turing machine, In fact a Turing machine is far more powerful than any real computer since it effectively has an infinite amount of memory.
The Church-Turing Thesis says essentially that:
Any real computation (algorithm) can be simulated by a Turing machine.
It should be noted that the Church Turing thesis is an axiom, it is the link between the mathematical world and the real world. No amount of mathematics can ever prove this thesis because it states a fact about the real world.
Thus Turing machines represent the ultimate model of computation, if a language is not recognizable by a Turing machine, NO algorithm can compute it.

3.3 Encoding

When we talk about Turing machines in a general sense, as we do now, it is unproductive to worry about the internal workings of the machine (specifying state tables etc.). We are more interested in a general description.
In general Turing machines may be given a general object for analysis. All that is required that the general object be rendered in a form that the Turing machine can interpret, a finite string of characters from a suitable alphabet.
By an *encoding* we mean some method of encoding an input into a suitable string for input into a Turing Machine. Note that any finite information may be encoded, we are not concerned with the method of encoding, merely that it can be done.
We denote an encoding of an object by placing it between angled brackets: <>
For example if we wish to design a Turing machine to decide some property of a graph, G, we must first encode the graph: $< G >$.
The encoding will consist of some way of describing G's vertices and edges in the input alphabet of the Turing machine.
We can now describe languages with respect to the encoded object:
$L_{eul} = \{< G > \mid G \text{ is a graph which has an Eulerian circuit } \}$,
$L_{ham} = \{< G > \mid G \text{ is a graph which has an Hamiltonian circuit } \}$,
Both of these languages are decidable.i.e. There exists a Turing machine, M, such that $L(M) = L_{eul}$, this machine is input the encoding of a particular graph G, $< G >$, and accepts if G has an Eulerian circuit, and rejects if G does not.
Any object with a finite description may be encoded in this way. In particular, note that every part of the definition of a Turing machine $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$ is finite.

4 The Halting Problem (5.4)

We now consider problems which are not decidable.
There are many important problems which are known to be recognisable but not undecidable. For example it turns out that the problem of software verification (verifying that a program works as specified) is undecidable.
First we turn to Russell's paradox, which first appeared at the beginning of the 20^{th} century when Bertrand Russell and Alfred Whitehead tried to axiomatise set theory.

4.1 Russell's Paradox

With the advent of axiomatic theory it seemed reasonable that it should be possible to write a book, which started with some basic axioms, and then derived all of mathematics from these axioms. At the beginning of this century Alfred Whitehead (1861 - 1947) and Betrand Russell (1872 - 1970) attempted to write just such a book, *Principia Mathematica,* which actually appeared in three volumes from 1910 to 1913. The scope of this work was incredible, the proof that $1 + 1 = 2$ does not appear until page 362.
However Russell and Whitehead ran into a problem, they kept running into inconsistencies. Every time they tried to fix an inconsistency it would pop up again elsewhere. Though they did eventually publish, the work was flawed in that it was incomplete, there were mathematical theorems it could not address from the fundamental axioms.
The basic inconsistency that they found is known as Russell's paradox. Russell provided the following simple puzzle, known as the barbers paradox, to exemplify the problem:

> In a certain town there is a male barber who shaves all those men, and only those men, who do not shave themselves.
> The question is, who shaves the barber?

If the barber does not shave himself, then he should, since he shaves all those men who do not shave themselves. On the other hand the barber *only* shaves such men, and hence cannot shave himself.
Sets can have other sets as members, e.g. $\{1, \{2, 3\}\}$.
What about a set that conatains itself as a member.
As an example consider the universal set of everything.
Since this set conatains everything it must contain itself.
While the property of self containment is unusual it is not, in and of itself, paradoxical.
We obtain a paradox when we consider 'the set of all sets which are not members of themselves'.

$$R = \{ \text{ Sets } A \,|\, A \notin A\}$$

The question is, is R a member of itself?
R cannot not be a member of itself, but it must be. This is known as Russell's paradox.
If we go back to the definition of set given earlier, we can actually find a way out of this dilemma.

> A *set* is a collection of objects, such that it is possible to state unequivocally whether any given object is in the set or not.

Since we have an object, R, for which it is not possible to state unequivocally $R \in R$ or $R \notin R$, we must conclude that, by this definition, R is not a set.

5 Gödel's Theorem

The final word comes from the Austrian mathematician Kurt Gödel (1906 - 1978).
If it is possible to prove two mutually contradictory statements from a set of axioms, then this set of axioms is called *inconsistent,*
If there exists a theorem which **cannot** be proved or disproved from a set of axioms, then this set of axioms is called *incomplete.*
Gödel's theorem (1931) may be succinctly stated as follows.

> Every formal axiomatic system, with a finite number of axioms, is either incomplete or inconsistent.

What is truly incredible about this is that Gödel managed to use a formal mathematical approach to show that mathematics can never be both complete and consistent.
See *Gödel, Escher, Bach* by D. Hofstadter for a more complete discussion of Gödel's theorem.
Of course in mathematics we eshew inconsistency and always go for incompleteness.

5.1 The Halting Problem

We now use Russels paradox to show that there is an undecidable language.
Since every part of the definition of a Turing machine is finite and we may encode any finite object we may encode one Turing machine to be read as input to another. Thus, given a Turing machine M, we may encode it, $< M >$, into a form which may be read by a second Turing machine N.
We may include the input string as part of the definition of M, thus a complete encoding of a Turing machine, M, would be $< M, w >$, where w is the input to M.
We now design a *Universal Turing machine*, U, which accepts as input a Turing machine and its input, $< M, w >$, and simulates the action of M on the input w.
In fact this is exactly how a computer works, a programming language provides a way of describing an algorithm, which by the Church-Turing thesis is equivalent to a description of a Turing-machine M, at run time the program is supplied with the input w, the computer then simulates M running on w.
We now consider the following language:
$L_{TM} = \{< M, w > \mid M \text{ is a Turing machine, and } M \text{ accepts } w\}$.

The Halting Problem Is there a Turing machine which *decides* L_{TM}?

Theorem 14 *L_{TM} is recognizable.*

Proof: Use a universal Turing Machine to simulate M with input w.
If M enters q_{accept} accept.
If M enters q_{reject} reject.
Thus the problem is whether M halts on every input w.

Theorem 15 *L_{TM} is undecidable.*

Proof: (By Contradiction.) Suppose that L_{TM} is decidable, thus there is a Turing machine H, which always halts, which recognizes L_{TM}.
We write

$$H(< M, w >) = \begin{cases} \textit{accept} & \text{if } M \text{ accepts } w \\ \textit{reject} & \text{if } M \text{ does not accept } w \end{cases}$$

Now we design a new Turing machine D, which uses H as a 'subroutine'. The input to D is the encoding of a Turing machine $< M >$. On input $< M >$, D runs H on $< M, < M >>$. i.e. H determines the outcome of running the Turing machine M on an encoding of itself. D then reverses the output from H:

$$D(< M >) = \begin{cases} \textit{accept} & \text{if } M \text{ does not accept } < M > \\ \textit{reject} & \text{if } M \text{ accepts } < M > \end{cases}$$

We now run D on itself to derive a contradiction:

$$D(< D >) = \begin{cases} \textit{accept} & \text{if } D \text{ does not accept } < D > \\ \textit{reject} & \text{if } D \text{ accepts } < D > \end{cases}$$

The statement D accepts $< D >$ indicates that on input $< D >$, D accepts, But it rejects, a contradiction. Thus L_{TM} is undecidable.
This means that no Turing machine can decide whether a second Turing machine will eventually halt.
Has it crashed, or is it just taking a long time?

6 Unrecognizable Languages (7.5)

We now show that there are languages which are not recognizable by any Turing machine. That is languages whose structure is not algorithmically soluble.

6.1 Cardinality of Sets

Definition 16

1. We say that two sets X and Y have the same cardinality if there exists a bijection $f : X \to Y$.
2. If $Y = \{1, 2, \ldots, n\}$, for some fixed $n \in \mathbf{Z}^+$, and there exists a bijection $f : X \to Y$, then we say that X is of size n, and write $|X| = n$.

Note that there exists a bijection $f : X \to Y$ if and only if there exists a bijection $g : Y \to X$.
What happens if we extend the first part of this definition to infinite (unbounded) sets?

Definition 17 *A set X is said to be countably infinite, or countable if it has the same cardinality as $\mathbf{Z}^+$. i.e. A set S is countable if and only if there exists a bijection $f : S \to \mathbb{Z}^+$.*

The first person to really consider the notion of infinity in this way was Georg Cantor (1845 - 1918). Cantor was born in St. Petersburg, Russia, but moved to Berlin when he was 11. Cantor studied mathematics at Zurich, and ended up teaching at Halle university in Germany. Cantor never gained the recognition he felt he deserved, and during his life his ideas were ridiculed by the mathematical community. This effected him deeply and he spent much of his life in and out of mental institutions, suffering repeated breakdowns. However by the time of his death in 1918 his ideas were becoming widely accepted and his genius started to be recognized.

Even Numbers

Consider the set of positive even numbers, $E = \{2, 4, 6, 8, \ldots\}$.
Consider $f : \mathbf{Z} \to E,\ f(n) = 2n$.
f is a bijection (Exercise)

Natural Numbers	1	2	3	4	5	...
Even Numbers	2	4	6	8	10	...

Thus the conclusion is that the number of natural numbers has the same cardinality as the positive even numbers, very strange. Thus the positive even numbers are countable.
This leads to strange phenomena, such as Hilbert's hotel:

> Hilbert's hotel is a hotel with an infinite number of rooms, numbered $1, 2, 3, \ldots$
> One day the hotel is full, with an infinite number of guests.
> That day an infinite number of buses arrive carrying an infinite number of new guests.
> "No problem", says the manager, "we can accommodate you all".
> How does the manager accommodate the new guests?
>
> For each of the current guests the manager moves the person in room i to room $2i$, thus freeing up an infinite number of rooms for the new guests.

Z

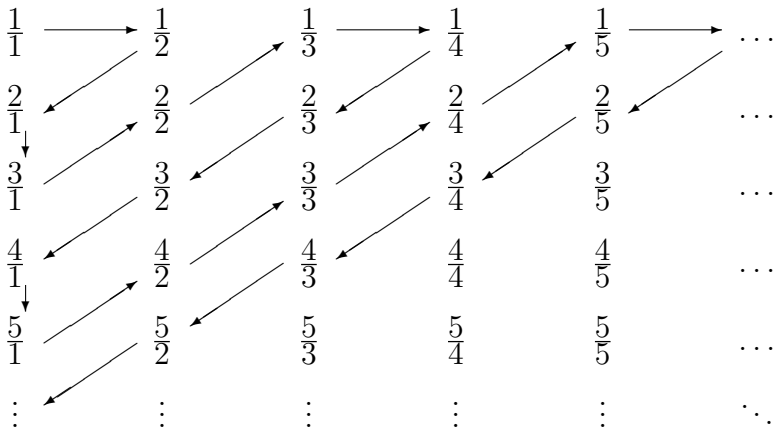
The integers are countable.
Consider the function, $f : \mathbf{Z}^+ \to \mathbf{Z},\ f(n) = \begin{cases} -\frac{n-1}{2} & \text{if } n \text{ is odd} \\ \frac{n}{2} & \text{if } n \text{ is even} \end{cases}$
The output of this function goes as follows:

n	1	2	3	4	5	...
$f(n)$	0	1	-1	2	-2	...

f is a bijection (Exercise)
Thus the integers have the same cardinality as the positive integers, they are countable.

Q

The rational numbers, $\mathbf{Q}$, are countable.
In order to count the rational numbers we create an infinite table on which to count, the denominator increases as we move to the right, and the numerator increases as we move down (see figure). This enumerates all possible values of numerator and denominator. We now count diagonally starting at the top left, every time we reach the top we move right, and every time we reach the left hand side we move down.



This assigns a unique natural number to each rational number.

n	1	2	3	4	5	6	...
$f(n)$	$\frac{1}{1}$	$\frac{1}{2}$	$\frac{2}{1}$	$\frac{3}{1}$	$\frac{2}{2}$	$\frac{1}{3}$	...

Thus the rational numbers are countable, i.e. they have the same cardinality as the natural numbers.

[0,1]
We now show that the set of real numbers between 0 and 1 is not countable.
This means that there is *no* one to one correspondence between this set and the set of natural numbers. We are thus faced with the problem of showing that something does not exist, always much harder than showing that something does.
The answer is Cantor's diagonalisation proof:

Theorem 18 *There is no bijection* $f : \mathbf{Z}^+ \to [0, 1]$.

Proof: (By contradiction)
Suppose not, that is suppose that there is a bijection $f : \mathbf{Z}^+ \to [0, 1]$.
That is, to each real number between 0 and 1 we can assign a unique positive integer.
If this were so we could take an infinitely large sheet of paper and write out the natural numbers, n corresponding to the real numbers x. An example of how part of such a table might look is shown below

n	x
1	$0.\underline{1}9867591435987253098615 32\ldots$
2	$0.6\underline{5}6987234502345879623 4509\ldots$
3	$0.29\underline{3}8745723450972345234 534\ldots$
4	$0.985\underline{4}9182734509123465987 64\ldots$
5	$0.19875\underline{2}3444098234734598 723\ldots$
6	$0.23418\underline{9}7123487912349876 123\ldots$
$\vdots$	$\vdots$ $\ddots$

Now we will create a real number between 0 and 1 which cannot be on the table.
On the ith row we identify the ith digit after the decimal point.
In this example this gives the digits $1, 5, 3, 4, 5, 9, \ldots$
We can consider this as a number between 0 and 1, namely $0.153459\ldots$
Now, for each identified digit we change it to something else, for example we might add 1, changing 9 to 0.
This gives a new number, in this case $0.264560\ldots$
This number, which lies between 0 and 1, cannot be on the list.
The reason is that it differs from the first number on the list in the first position after the decimal point, from the second number on the list in the second position after the decimal point, from the third number on the list in the third position after the decimal point, and so on.
In general it differs from the ith number on the list in the ith position after the decimal point, since we started with that digit and then changed it.
Thus we have created a number between 0 and 1 which does not have a natural number assigned to it, if it did it would be on the list. □
This shows that the number of real numbers between 0 and 1 is not countable, this is a different infinity!
This infinity is called the continuum.
Note that the above proof works just as well if the numbers are represented in binary decimal, $0.001001110\ldots$ etc.

6.2 Application to Turing Machines

Theorem 19 *For any finite alphabet Σ, Σ^* is countable.*

Proof: List out the elements of Σ^* in some order, and count them. □
For example if $\Sigma = \{0, 1\}$

$$\begin{array}{lcl} \Sigma^* & = & \{\ \underset{0}{\epsilon},\ \underset{1}{0},\ \underset{2}{1},\ \underset{3}{01},\ \underset{4}{10},\ \underset{5}{000},\ \underset{6}{001},\ \underset{7}{010},\ \underset{8}{011},\ \underset{9}{100},\ \underset{10}{101},\ \underset{11}{111},\ \ldots\ \} \end{array}$$

Theorem 20 *The number of Turing machines is countable*

Proof: As previously noted the definition of a Turing machine is finite.
$M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$
So, every Turing machine has some finite encoding $< M > \subseteq \Sigma^*$. Where Σ is some finite input alphabet.
But as already noted Σ^* is countable.
Let $\mathcal{L}_\Sigma = \{L \mid L \text{ is a language over} \Sigma\} = \mathcal{P}(\Sigma^*) =$ The set of all languages over Σ.

Theorem 21 *For any finite alphabet Σ, $\mathcal{L}_\Sigma$ is uncountable.*
(The power set of a countable set is uncountable.)

Proof: For each language $L \in \mathcal{L}_\Sigma$ ($L \subseteq \Sigma^*$) we define the Characteristic sequence of L, χ_L as follows.

List the elements of Σ^* in some fixed order. We define the i^{th} element of the characteristic sequence of L by

$$\chi_L(i) = \begin{cases} 1 & L \text{ contains the } i^{\text{th}} \text{ element of } \Sigma^* \\ 0 & \text{otherwise} \end{cases}$$

For example if $\Sigma = \{0, 1\}$, $L = 0(0 \vee 1)^*$ then χ_L:

$$\begin{array}{lcccccccccccccc} \Sigma^* & = & \{ & \epsilon, & 0, & 1, & 01, & 10, & 000, & 001, & 010, & 011, & 100, & 101, & 111, & \ldots & \} \\ L & = & \{ & & 0, & & 01, & & 000, & 001, & 010, & 011, & & & & \ldots & \} \\ \chi_L & = & & 0 & 1 & 0 & 1 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & \ldots & \end{array}$$

The sequence χ_L may be interpreted as a decimal number between 0 and 1 in binary, ($0.010101111000\ldots$ in this case). So by the Cantor's diagonalisation argument above the set of sequences $\{\chi_L \mid L \in \mathcal{L}\}$ is uncountable

Note that each $L \in \mathcal{L}$ defines a unique sequence, and each sequence defines a unique $L \in \mathcal{L}$, so the function $\chi : \mathcal{L} \to [0, 1]$ is a bijection. Thus $|\mathcal{L}| = |[0, 1]|$, which is not countable.

Note that though we have proved that an unrecognizable language must exist we cannot explicitly give one. Indeed, how would we specify such a language? The only way to specify which strings are in an infinite language and which are not is to give some algorithm or set of rules. But by the Church Turing thesis the existence of an algorithm to find a language is equivalent to the existence of a Turing machine to recognize it.

Corollary 22 *The set of problems is larger than the set of (algorithmic) solutions.*

Or: There are more questions than there are algorithmic answers.
Or: There are some things computers just can't do!

7 Time Complexity

Up to this point we have been concerned with whether it is *possible* to compute a language. In this section we will briefly consider the time taken for a decidable language

Definition 23 *Given a Turing Machine, M, which* **decides** *a language $L \subseteq \Sigma^*$.*

1. *The time taken for M to process a string w, $T_M(w)$, is the number of steps M goes through before halting on input w.*
2. *The time complexity of M,*
$$T_M(n) = \{T_M(w) \mid |w| = n\}.$$

Note that the time complexity of a machine M, is a function of the length of the input string, and that it considers the worst case.

By the Church Turing thesis any algorithm may be implemented by some Turing Machine M. We take the complexity of an algorithm to be the time complexity of the most efficient Turing machine which implements the algorithm.

Example 24

1. Consider the Turing machine M which decides $(0 \vee 1)0 = \{x \in \{0,1\}^* \mid x \text{ ends in } 0\}$.

   M scans to the right end of the input string and accepts if the last symbol is a 0.

   Scanning across an input string w, where $|w| = n$, takes n steps.

   Thus $T_M(n) = n$.

2. Consider the example given above which decides the language $L = \{x \in \{0\}^* \mid x = 0^{2^m}, m \in \mathbf{N}\}$, by implementing the following algorithm:

   Scan right along the tape crossing off every other 0, this halves the number of 0's.
   If there is only one 0, accept.
   If the number of 0's is odd reject. (Note the parity of 0's can be recorded by state.)

   Scan back to the left hand end of the string.

   Repeat.

   The original input string has length $n = 2^m$, for some m.
   Thus each scan across the input string takes $n = 2^m$ steps.
   We must do $m = \log_2 n$ such scans.

   So $T_M(n) = n \log_2 n$

3. The method of encoding can make a difference to the complexity, since it can shorten or lengthen the input. However there is usually a cutoff beyond which any gains are relatively minimal.

   Consider a Turing Machine M which implements the following empty loop:

   input m
   do m steps (for($i = 0$, $i < m$; i++);) Halt

   Suppose m is encoded in unary (i.e. $\Sigma = \{0\}$ and m is represented by 0^m).
   Then $T_M(n) = n$.

   Now suppose that m is encoded in binary (i.e. $\Sigma = \{0,1\}$ and m is represented in binary).
   Now the number of bits needed to represent m is roughly $\log_2 m$.
   So the length of the input is $n = \log_2 m$, but the machine does $m = 2^n$ steps.
   So $T_M(n) = 2^n$.

   Now suppose that m is encoded in trinary (base 3) (i.e. $\Sigma = \{0,1,2\}$ and m is represented in trinary).
   A similar calculation to that above gives $T_M(n) = 3^n = 2^{an} = 2^a \cdot 2^n$, where $a = \log_2 3$.

   Thus we see that if m is represented in unary the running time is linear, whereas if m is represented in binary the running time is exponential. A very larger relative difference in running times.

   On the other hand the difference between a binary and trinary representation of m takes us from 2^n to 3^n, both are exponential and differ only by a constant factor. Thus relatively the difference is minimal, unlike the change from unary to binary.

   Note that in general if m is reperesented in b-ary $T_M(n) = b^n$

Note When considering complexity we are note generally interested in the minutae of the computation. But rather in the order of magnitude of the time taken. We thus represent running times using the big O order notation.

Definition 25

1. *We say that an algorithm (or language) is Polynomial Time if it can be decided by a (deterministic) Turing Machine M with $T_M(n)$ a polynomial. i.e. $T_M(n) = O(n^a)$ for some fixed $a \in \mathbb{N}^+$.*

2. *The class Polynomial is the class of all algorithms which are polynomial time.*
   $P = \{L \mid \exists$ a deterministic Turing Machine M with $T_M(n) = O(n^a)$ for some fixed $a \in \mathbf{N}^+$ $\}$

3. *We say that an algorithm (or language) is Nondeterministic Polynomial Time if it can be decided by a* nondeterministic *Turing Machine M with $T_M(n)$ a polynomial. i.e. $T_M(n) = O(n^a)$ for some fixed $a \in \mathbf{N}^+$.*

4. *The class Nondeterministic Polynomial is the class of all algorithms which are nondeterministic polynomial time. $NP = \{L \mid \exists$ a nondeterministic Turing Machine M with $T_M(n) = O(n^a)$ for some fixed $a \in \mathbf{N}^+$ $\}$*

One of the fundamental questions of Computer Science is does $P = NP$?
There are algorithms which are in NP, but for which no deterministic polynomial time algorithm is known. (eg. Hamiltonian Circuit Problem.)
Since we are unable to actually build a nondeterministic machine we would very much like to find such an algorithm if it exists, or prove that no such algorithm can exist.