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.

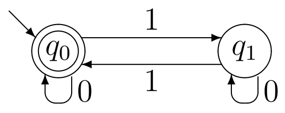
Example 3

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

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.

1.1 Nondeterminism

We note that for an FSA there is exactly one arc out of each state for each symbol of the alphabet. We relax this rule to get nondeterminism. Nondeterminism may be thought of as a kind of parallel processing. Specifically, a *Nondeterministic Finite state autoamata* (NFA) is exactly like an FSA except we can have:

Missing Arcs. In an NFA there may be no arc for a given symbol from a particular state, representing a computational dead end.

Multiple Arcs. In an NFA's there may be many different arcs leaving a state, each leading to a different possible computational path.

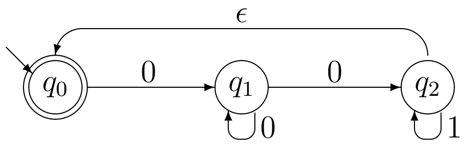
ϵ−arcs. In an NFA's there may be special arcs called ϵ−arcs between states. These arcs represent a branch, computation remains in the initial state and also propagates to the new state.

At each step an NFA is in multiple states, and takes every relevant arc to the next set of states.

Definition 6 *A* nondeterministic Finite State Automata *(NFA) is a 5-tuple* $(Q, \Sigma, \delta, q_0, F)$*.* Q*,* Σ*,* q_0 *and* F *are the same as for an FSA. The state function now maps sets of states to sets of states:* $\delta : \mathcal{P}(Q) \times (\{\epsilon\} \cup \Sigma) \to \mathcal{P}(Q)$ *where* $\mathcal{P}(Q)$ *is the power set of* Q*.*

As with FSAs the machine is provided an *input string* which it reads sequentially, one character at a time. At any point in the computation the machine may be in multiple states. Thus unlike FSAs which are in a single state, NFAs are in a set of states $A \subseteq Q$. Initially machine is in only the start state, so $A = \{q_0\}$. If the machine is in the set of states A and it reads the character a from the input string it will be in the set $\delta(A, a)$. If there are any ϵ−arcs from any of the elements of $\delta(A, a)$ then these are added to the set of active states.
If after reading the final character of the input string one of the states that the machine is in is 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.
The definitions above of $L(M)$, equivalence, eventual state (δ^*) etc. carry over to NFA's.
In the state diagram we can consider a transition corresponding to reading character a to be taking every arc labeled a from every state that the machine is currently in.
We can also consider the *computation tree* associated with the action of an NFA on a given input string.

Example 7



This machine accepts the empty string.
If the input string is '1', the computation path dies and the machine is in no state, and hence rejects.
In order to stay alive the machine must read a 0 as the first character. However, it is now in a non accepting state (q_1).
Upon reading another 0 the machine takes two arcs, the loop back to q_1 and the arc on to q_2, once in q_2 the ϵ−arc to q_0 is automatically taken and so the machine is in all three states $\{q_0, q_1, q_2\}$.
Since one of these states (q_0) is accepting the string 00 is accepted.
If a 0 is now read the machine takes all the 0 labelled arcs from all the states that it is in $q_0 \to q_1$, $q_1 \to q_2$, ϵ−arc to q_0 and so it remains in all states $\{q_0, q_1, q_2\}$.
On the other hand, if a 1 is now read q_0 and q_1 die and only q_2 survives via the loop. The ϵ−arc is immediately taken to q_0, so the machine is in states $\{q_0, q_2\}$. q_0 is accepting, so the string is accepted.
$L(M) = (00^*01^*)^*$ = Begins with 0 and has an even number of 0's.

1.2 Equivalence of Nondeterminism

Definition 8 *Given a set of states A, the ϵ closure of A, $E(A)$, is the set of all states which are either in A or can be reached from a state in A by a path consisting entirely of $\epsilon-$arcs. We define $E(\emptyset) = \emptyset$.*

Theorem 9

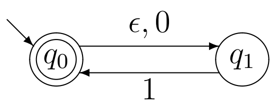
1. $A \subseteq E(A)$.
2. *If* $A_1 \subseteq A_2$ *then* $E(A_1) \subseteq E(A_2)$.
3. $E(E(A)) = E(A)$.

Note that if an NFA is in a set of states A and character b is read the machine will be in states $E(\delta(A, b))$.

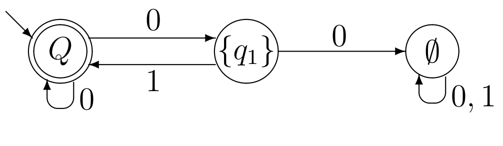
Theorem 10 *Every NFA has an equivalent FSA.*

Proof: Suppose $M = (Q, \Sigma, \delta, q_0, F)$ is an NFA, we will construct an equivalent FSA $N = (P, \Sigma, \gamma, p_0, G)$. Note that since we wish to have $L(M) = L(N)$ the input alphabets are the same.
$P = \{E(\{A\}) \mid A \subseteq Q\}$ = the set of all ϵ closed subsets of Q.
$p_0 = E(\{q_0\})$.
$G = \{p \in P \mid p \cap F \neq \emptyset\}$. (Note p is a subset of Q.)
For $p \in P$ and $a \in \Sigma$, $\gamma(p, a) = \{q \in Q \mid \exists\, q' \in p$ such that $q \in E(\delta(q', a))\}$ = the set of states reachable in M from a state in p on reading a.
The idea is that N has a state for each set of states that M could be in and an arc between two states, p and p' labelled a if M moves from the set of states p to the set of states p' on reading a.
Thus while NFAs are generally easier to design and work with, every language accepted by an NFA is accepted by a DFA, so NFAs do not extend the range of languages we have to deal with.

Example 11
Find an FSA which is equivalent to the NFA below:



$$
\begin{array}{lclcl}
E(\{q_0\}) & = & \{q_0, q_1\}, & & \\
E(\delta(\{q_0, q_1\}, 0)) & = & E(\{q_1\}) & = & \{q_1\}, \\
E(\delta(\{q_0, q_1\}, 1)) & = & E(\{q_0, q_1\}) & = & \{q_0, q_1\}, \\
E(\delta(\{q_1\}, 0)) & = & E(\emptyset) & = & \emptyset, \\
E(\delta(\{q_1\}, 1)) & = & E(\{q_0\}) & = & \{q_0, q_1\}
\end{array}
$$



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 12 *The language $L = \{a^n b^n \mid n \in \mathbb{Z}\}$ is not accepted by any FSA(or NFA).*

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 $\delta^*(q, a^r) = q$.

$$\begin{tikzcd} q_0 \arrow[r, "a^p"] & q \arrow[loop below, "a^r"] \arrow[r, "a^q b^n"] & q_a \end{tikzcd}$$

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 13 (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 $|w| > p$, 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 $|w| > p$, ideally the first p characters of w are the same (a say).
Since $|w| > p$ 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 14

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 $w = 0^p 1^p$, $|w| > p$, 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 $w = 0^p10^p$, $|w| > p$, 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 Grammars

Consider the following excerpt from the (somewhat simplified) definition for C function declaration syntax:

```
<type> <identifier> (<identifier list>) {<statement list>}
<type> := char | int | short | long | float | double | <type> *
<identifier> := <letter> | <letter><alphanumeric string>
<alphanumeric string> := <alphanumeric> | <alphanumeric><alphanumeric string>
<alphanumeric> := <digit> | <letter>
<digit> := 0 | 1 | 2 | ... | 9
<letter> := a | ... | z | A | ... | Z
<identifier list> := <identifier> | <identifier>, <identifier list>
                                        ⋮
```

Here `|` is "or".
This definition starts with an initial rule, involving *variables* (indicated by `<` ... `>`) and *terminals* (`0 - 9, A - Z, a - z, (, ), {, }, *`).
The variables are then further broken down into more terminals and variables.
The terminals are members of an alphabet; in this case ASCII, together with the words `char, int, short, long, float, double`.
The definition thus consists of a set of *production rules* which tell us how to form new strings of terminals and variables out of old. This is what is known as a *Grammar*
This form (using `< >` for variables, `:=` for productions and `|` for "or") is called Bakus-Naur Form.

Definition 15 *A (phrase structure) Grammar G is a 4-tuple (Σ, V, S, P), where:*

1. *Σ is an alphabet of terminal symbols.*
2. *V is a finite set of non terminal, or variable, symbols.*
3. *$S \in V$ is a start symbol.*

4. *P is a set of production rules of the form $\alpha \rightarrow \beta$ where $\alpha, \beta \in (V \cup \Sigma)^*$, and α contains at least one non terminal symbol.*

In general productions take a string (on the left) containing a non terminal symbol and tell us how that symbol may be expanded.

3.1 Notation

In general we use the following notation:

1. Upper case letters from the beginning of the alphabet to denote a single variable from V. Note the special symbol S is also a variable. $A, B, C, D, E \in V$.
2. Lower case letters from the beginning of the alphabet to denote a single terminal. $a, b, c, d, \ldots \in \Sigma$.
3. Lower case letters from the end of the alphabet to denote strings of terminals. $u, v, w, x, y, z \in \Sigma^*$.
4. Upper case letters from the end of the alphabet to denote a single symbol, either variable or terminal. $U, V, W, X, Y, Z \in V \cup \Sigma$
5. Lower case Greek letters form the beginning of the alphabet to denote strings consisting of terminals and$\alpha, \beta, \gamma, \delta \in (V \cup \Sigma)^*$

Example 16
$H = (\Sigma, V, S, P)$, $V = \{S\}$, $\Sigma = \{a, b\}$
Productions:

$$\begin{array}{rcl} S & \rightarrow & aSb \\ S & \rightarrow & \epsilon \end{array}$$

Definition 17 *Given a grammar G*

1. *If $\alpha \rightarrow \beta$ is a production rule of G we say that β is directly derivable from α.*
2. *$\alpha \overset{n}{\Rightarrow} \beta$ means that there are exactly n productions of G which will take us from α to β.*
   i.e. $\exists\, \alpha_0, \alpha_1, \ldots, \alpha_m \in (V \cup \Sigma)^*$ with $\alpha_0 = \alpha$ and $\alpha_m = \beta$ such that for each i from 1 to n, α_{i+1} is directly derivable from α_i. i.e. $\alpha_i \rightarrow \alpha_{i+1}$ is a production rule of G.
3. *$\alpha \overset{*}{\Rightarrow} \beta$ means $\alpha \overset{n}{\Rightarrow} \beta$ for some $n \in \mathbf{N}^+$. We say that β is derivable from α.*
4. *The language generated by G, $L(G)$, is the set of all strings consisting of only terminals which are derivable from the start variable S of G.*
   i.e. $L(G) = \{x \in \Sigma^* \mid S \overset{*}{\Rightarrow} x\}$

Example 18 *Continuing with H above.*

- *aSb is directly derivable from S.*
- *$aaSbb$ is derivable from S, by the sequence of productions $S \rightarrow aSb \rightarrow aaSbb$.*

- *So $S \overset{2}{\Rightarrow} aaSbb$.*
- *$S \overset{3}{\Rightarrow} aabb$, by the sequence of productions $S \to aSb \to aaSbb \to aabb$.*
- *$L(G) = a^n b^n$*

As we have defined grammars any type of production rule is allowed, we classify grammars by the type of productions which are allowed.

Definition 19 *We define the following hierarchy of grammars, known as the Chomsky Hierarchy:*

1. The set of all grammars is called Unrestricted.
2. If every production is of the form $\alpha A \beta \to \gamma$ $(\alpha, \beta, \gamma \in (\Sigma \cup V)^*, A \in V)$ we call the grammar Context Sensitive.

   In this case the productions for variables may depend on the surrounding context (α and β). For example $bA \to bab$ means that we may only derive ab from A if A is preceded by a b.
3. If every production is of the form $A \to \alpha,$ $(A \in V, \alpha \in (\Sigma \cup V)^*)$ we call the grammar Context Free.

   In this case productions of variables do not depend on context.Each variable may be expanded to a string of variables and terminals.The example H above is a context free grammar.Backus-Naur form always gives a context free grammar.
4. If every production is of the form $A \to a,$ or $A \to aB,$ or $A \to \epsilon$ $(A, B \in V, a \in \Sigma)$ we call the grammar Regular

Theorem 20 *If G is a regular grammar then $L(G)$ is a regular language.*

Definition 21

1. *Given a language L, it is called context free if there exists a context free grammar G such that $L = L(G)$.*
2. *Given a language L, it is called context sensitive if there exists a context sensitive grammar G such that $L = L(G)$.*
3. *Given a language L, it is called unrestricted if there exists an unrestricted grammar G such that $L = L(G)$.*

Note These languages do form a hierarchy in the sense that
{ Regular languages } $\subseteq$ { Context free languages } $\subseteq$ { Context sensitive languages } $\subseteq$ { Unrestricted languages }.

Definition 22 *Given two grammars G and G' they are called equivalent if they generate the same language, i.e. $L(G) = L(G')$.*

Note In general it is hard to find explicit examples of unrestricted grammars which are not equivalent to context sensitive grammars, however it is possible to prove that such grammars must exist.

Example 23

1. (Regular Grammar) $\Sigma = \{a, b\}$, $V = \{S, A, B\}$
   Productions:
   $$\begin{array}{lllll} 1. & S & \rightarrow & aA \\ 2. & A & \rightarrow & bB \\ 3. & B & \rightarrow & aA \end{array}$$
   $L(G) = a(ab)^*$

2. (Context Free Grammar) $\Sigma = \{a, b\}$, $V = \{S, A, B\}$
   Productions:
   $$\begin{array}{lllll} 1. & S & \rightarrow & aB \mid bA \\ 2. & B & \rightarrow & aBB \\ 3. & B & \rightarrow & bS \\ 4. & B & \rightarrow & b \\ 5. & A & \rightarrow & bAA \\ 6. & A & \rightarrow & aS \\ 7. & A & \rightarrow & a \end{array}$$
   $L(G) = \{x \in \{a, b\}^* \mid x$ has an equal number of a's and b's (in any order) $\}$

3. (Context Sensitive Grammar) $\Sigma = \{a, b, c\}$, $V = \{S, A, B, C, D\}$
   Productions:
   $$\begin{array}{lllll} 1. & S & \rightarrow & aAB \mid aB \\ 2. & A & \rightarrow & aAC \mid aC \\ 3. & B & \rightarrow & Dc \\ 4. & D & \rightarrow & b \\ 5. & Cc & \rightarrow & Dcc \\ 6. & CD & \rightarrow & DC \end{array}$$
   consider the following derivation:
   $$S \Rightarrow aAB \Rightarrow aaACB \Rightarrow \overbrace{aaa}^{n}\overbrace{CC}^{n-1} B \Rightarrow \overbrace{aaa}^{n}\overbrace{CC}^{n-1} Dc \Rightarrow \overbrace{aaa}^{n} D \overbrace{CC}^{n-1} c \Rightarrow \overbrace{aaa}^{n} D \overbrace{C}^{n-2} Dcc \Rightarrow$$
   $$\overbrace{aaa}^{n}\overbrace{DD}^{n-1} C \overbrace{cc}^{n-1} \Rightarrow \overbrace{aaa}^{n}\overbrace{DDD}^{n}\overbrace{ccc}^{n} \Rightarrow \overbrace{aaa}^{n}\overbrace{bbb}^{n}\overbrace{ccc}^{n}$$
   $L(G) = \{x \in \{a, b\}^* \mid x = a^n b^n c^n, n \in \mathbf{N}^+\}$

4. (Context Sensitive Grammar) $\Sigma = \{a\}$, $V = \{S, A, B, C, D, E\}$
   Productions:
   $$\begin{array}{lllll} 1. & S & \rightarrow & ACaB \\ 2. & Ca & \rightarrow & aaC \\ 3. & CB & \rightarrow & DB \\ 4. & aD & \rightarrow & Da \\ 5. & AD & \rightarrow & AC \\ 6. & CB & \rightarrow & E \\ 7. & aE & \rightarrow & Ea \\ 8. & AE & \rightarrow & \epsilon \end{array}$$
   A and B mark the ends of the string (Rule 1).
   C moves right doubling the number of a's in the string (Rule 2).
   When C reaches the right hand end it changes to a D (Rule 3).

D moves left (Rule 4) and reinstitutes C at the left hand end of the string (Rule 5).
Repeating this C doubling n times gives a string of the form $Aa^{2^n}CB$.
Use E to erase B (Rule 6), move E left (Rule 7), and erase A and E (Rule 8).
$L(G) = \{x \in \{a\}^* \mid x = a^{2^n}, n \in \mathbf{N}^+\}$

We saw that regular languages are the class of languages that are accepted by FSA's. In a similar manner for each class of Language defined by a class of grammar (Regular, Context free, Context Sensitive) there is a "natural" generalisation of FSAs which accepts that class of language.
Regular languages are accepted by FSAs, we saw that the problem with FSAs was that they had bounded memory. We thus may add a (FIFO) stack to an FSA to obtain a *Pushdown Automata* (PDA). Notee that this stack is of unbounded size.

Theorem 24 *A language is accepted by a PDA if and only if it is context sensitive.*

A Linear Bounded Automata (LBA) is a Turing Machine (see below) with a linear bound on the size of the tape.

Theorem 25 *A language is accepted by an LBA if and only if it is context free.*

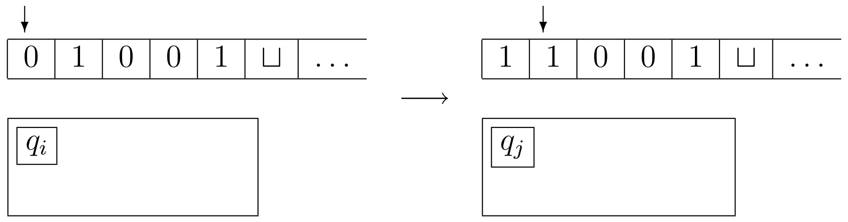
We thus see that there is a correspondence between computational power and the language that can be computed. Each language represents a problem and each machine an algorithmic solution. Rather than investigate these types of machines in detail we instead consider the most powerful model of computation known, the Turing Machine.

4 Turing Machines

We have seen how there is a correspondence between types of languages and the types of machines that accept them. We can find a class of machines that recognizes context sensitive grammars (push-down automata (PDA)) and a class that accept context sensitive ones (Linear Bounded Automata (LBA)). 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 26 (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 (⊔). 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.)
Note In this definition the Finite State Control of the Turing machine is deterministic, i.e. a DFA.

Example 27

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:

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

Definition 28

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 29

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 $L = 0^n1^n$.

   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.

$$\begin{array}{lclclclcl}
q_0 0011\sqcup & \longrightarrow & \# q_1 011\sqcup & \longrightarrow & \#0 q_1 11\sqcup & \longrightarrow & \#01 q_1 1\sqcup & \longrightarrow \\
\#011 q_1 \sqcup & \longrightarrow & \#01 q_2 1\sqcup & \longrightarrow & \#0 q_3 1\#\sqcup & \longrightarrow & \# q_3 01\#\sqcup & \longrightarrow \\
q_3 \#01\#\sqcup & \longrightarrow & \# q_4 01\#\sqcup & \longrightarrow & \#\# q_1 1\#\sqcup & \longrightarrow & \#\#1 q_1 \#\sqcup & \longrightarrow \\
\#\# q_2 1\#\sqcup & \longrightarrow & \# q_3 \#\#\#\sqcup & \longrightarrow & \#\# q_4 \#\#\sqcup & \longrightarrow & \text{Accept}
\end{array}$$

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			

4.1 Variations

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

4.1.1 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'.

4.1.2 Nondeterministic Turing Machines

One obvious variation is to allow the finite state control for the Turing machine to be nondeterministic.
In this case the transition function becomes

$$\delta : Q \times \Gamma \longrightarrow \mathcal{P}(Q \times \Gamma \times \{L, R\}^k)$$

The processing becomes nondeterministic.
Note that each computation path will proceed with different tape contents.

Theorem 30 *If a language is recognized by some nondeterministic Turing machine M, then it is recognized by some deterministic Turing machine M'.*

4.1.3 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 31 *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.

4.2 The Chomsky Hierarchy and Turing Machines

We would like a characterization of the languages accepted by Turing machines. Recall the Chomsky hierarchy:
Regular $\subseteq$ Context Free $\subseteq$ Context Sensitive $\subseteq$ Unrestricted.

Theorem 32 *A Language is* recognized *by some Turing machine if and only if it is generated by an unrestricted Grammar.*

We would also like to have a characterization of the languages which are Turing decidable, unfortunately no such characterization is known.

Theorem 33 *Every Context free Language is Decidable.*

Theorem 34 *Every Context Sensitive Language is Decidable.*

Thus showing that a language is generated by an unrestricted grammar, which is not context sensitive, is equivalent to solving the following problem:

The Halting Problem Given a Turing machine M does M halt on every input w?

4.3 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.

4.4 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:
$L_{eul} = \{< G > \mid G$ is a graph which has an Eulerian circuit $\}$,
$L_{ham} = \{< G > \mid G$ 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.

5 The Halting Problem (5.4)

We first consider algorithms which are not decidable.
There are many important problems which are known to be recognisable but not undecidable.
For example the problem of software verification (verifying that a program works as specified) is undecidable.
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:
$L_{TM} = \{< M, w > \mid M$ is a Turing machine, and M accepts $w\}$.

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

Theorem 35 *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 36 *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 (Review)

Definition 37

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 $f : Y \to X$.
What happens if we extend the first part of this definition to infinite (unbounded) sets?

Definition 38 *A set X is said to be countably infinite, or countable if it has the same cardinality as $\mathbf{Z}^+$.*

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,$f : \mathbf{Z}^+ \to \mathbf{Z},\ f(n) = \begin{cases} -\dfrac{n-1}{2} & \text{if } n \text{ is odd} \\ \dfrac{n}{2} & \text{if } n \text{ is even} \end{cases}$
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.

$$\begin{tikzcd}
\frac{1}{1} \arrow[r] & \frac{1}{2} \arrow[dl] & \frac{1}{3} \arrow[r] & \frac{1}{4} \arrow[dl] & \frac{1}{5} \arrow[r] & \cdots \arrow[dl] \\
\frac{2}{1} \arrow[d] & \frac{2}{2} \arrow[ur] & \frac{2}{3} \arrow[dl] & \frac{2}{4} \arrow[ur] & \frac{2}{5} \arrow[dl] & \cdots \\
\frac{3}{1} \arrow[ur] & \frac{3}{2} \arrow[dl] & \frac{3}{3} \arrow[ur] & \frac{3}{4} \arrow[dl] & \frac{3}{5} & \cdots \\
\frac{4}{1} \arrow[d] & \frac{4}{2} \arrow[ur] & \frac{4}{3} \arrow[dl] & \frac{4}{4} & \frac{4}{5} & \cdots \\
\frac{5}{1} \arrow[ur] & \frac{5}{2} \arrow[dl] & \frac{5}{3} & \frac{5}{4} & \frac{5}{5} & \cdots \\
\vdots & \vdots & \vdots & \vdots & \vdots & \ddots
\end{tikzcd}$$

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 39 *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	$0.\underline{1}98675914359872530986153 2\ldots$
2	$0.6\underline{5}5698723450234587962345 09\ldots$
3	$0.29\underline{3}874572345097234523453 4\ldots$
4	$0.985\underline{4}91827345091234659876 4\ldots$
5	$0.1987\underline{5}23444098234734598723\ldots$
6	$0.23418\underline{9}7123487912349876123\ldots$
$\vdots$	$\vdots \quad \ddots$

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 <u>continuum.</u>
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 40 *For any finite alphabet Σ, Σ^* is countable.*

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

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

Theorem 41 *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 42 *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 <u>Characteristic sequence</u> 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:

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

The sequence χ_L may be interpreted as a decimal number between 0 and 1 in binary, (0.010101111000... 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 $\chi : \mathcal{L} \longrightarrow [0,1]$ 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 43 *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 44 *Given a Turing Machine, M, which* **decides** *a language $L \subseteq \Sigma^*$.*

1. *The <u>time taken</u> 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 <u>time complexity</u> 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 45

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 46

1. *We say that an algorithm (or language) is* <u>*Polynomial Time*</u> *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* <u>*Polynomial*</u> *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* <u>*Nondeterministic Polynomial Time*</u> *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* <u>*Nondeterministic Polynomial*</u> *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.