1 Problems and Algorithms

In this section we will establish what we mean by a \textit{solution} to a computational \textit{problem}. This is the basis of the mathematical theory of computation.

1.1 Problems

First of all, what exactly do we mean by a “computational problem”? In general a \textit{problem} is specified by some information we are given (called the \textit{input}) and some information we want to compute from it (the \textit{output}). By a \textit{solution} to a problem, we mean an algorithm for starting with the input, and producing the required output. (We shall consider later what we mean by an “algorithm”!)

Example (The Travelling Salesman Problem). The input is a list of cities, and a table showing the shortest road distance between each pair. The required output is a list of the same cities \textit{in the order which minimises the distance travelled when visiting them in sequence}.

Example (Factorisation). The input is a positive integer \( n \). The required output is a sequence of prime numbers which multiply to give \( n \).

Example (Primality Testing). Here the input is again a positive integer \( n \). This time the required output is \textbf{YES} if \( n \) is prime, and \textbf{NO} if it is not.

Problems like the last one, where the output is a simple \textbf{YES} or \textbf{NO}, are called \textit{decision problems}. They play a special role in complexity theory.

Exercise. What are the input and output for the \textit{greatest common divisor} problem mentioned in the introduction? Is this a decision problem?

Exercise. Try to think of some more examples of computational problems, which you have met in your previous studies.

1.2 Languages and Codings

In practice, we need to know not just what the input and output are as abstract objects; we also need a concrete way in which they can be \textit{described} or \textit{coded}. In everyday mathematics we most often describe things using \textit{sequences of symbols}, and this suffices for computation too.

Definition. An \textit{alphabet} \( A \) is a finite set of symbols. A \textit{word} over the alphabet \( A \) is a finite sequence of (zero or more) symbols from \( A \); we write \( A^* \) for the set of all words over \( A \). A \textit{language} over \( A \) is a set of words over \( A \), that is, a subset of \( A^* \).

Definition. The number of symbols in a word \( w \) is called the \textit{length} of \( w \) and written \( |w| \). The unique word of length 0 is called the \textit{empty word} and written \( \epsilon \).

Example. The Roman alphabet \( \{a, b, \ldots, z\} \) forms an alphabet. The words over this alphabet include English words such as “\textit{cat}” (of length 3) and “\textit{redraft}” (of length 7) but also meaningless strings of letters like “\textit{gujdhks}”. The set of all English words (ignoring apostrophes and the like!) forms one language over the Roman alphabet. Another is the language PALINDROMES of all words which read the same backwards and forwards (like “\textit{pop}”, “\textit{redivider}” and “\textit{abcba}”).

Example. The set \( \{0, 1\} \) forms an alphabet. Some of the words over this alphabet (those which either are 0 or begin with a 1) can be seen as representing natural numbers in \textit{binary notation}. We say that these words form a \textit{coding} of the natural numbers. An example of a language is the set \textit{PRIMES} of words which represent prime numbers.

Definition. To each language \( L \subseteq A^* \) we can associate a decision problem: the input is a word \( w \) over \( A \). The required output is \textbf{YES} if \( w \in L \) and \textbf{NO} if \( w \notin L \). This problem is called the \textit{membership problem for} \( L \).
**Example.** The membership problem for the language PRIMES is essentially the same as the primality testing problem from Section 1.1. The only difference is that the input is not an abstract integer, but rather a *coding* of the integer as a binary number (a word over the alphabet \( \{0, 1\} \)).

### 1.3 Turing Machines

We now have a pretty good idea what a “problem” is, and we know that a “solution” is an “algorithm” to solve a problem. But what exactly do we mean by an “algorithm”? Essentially, an algorithm is a set of instructions which could be carried out by a mechanical device, in other words a *computer*\(^1\). There are lots of different sorts of computers, but there is a sense in which they are all fundamentally equivalent. We want a simple theoretical model of a computer which captures the essence of computation, while avoiding the technicalities of semiconductor electronics.

Such a model was proposed by Alan Turing (late of this University) before real computers even existed. And despite all the technical advances of the last fifty years, it is still taken to be a faithful model of computation.

Precisely, a (1-tape, deterministic) *Turing Machine* (or *TM*) is a machine which has a two-way infinite storage tape, divided into squares like this:

```
... B B B B B B B B ...
```

At any one moment, each square can either be blank, or can contain a single symbol from some alphabet. The machine has a *head* which at any one moment is pointing at one of the squares on the tape. The machine also has a finite set \( \{s_0, s_1, \ldots, s_m\} \) of *states*; at any one moment, it is in one of these states.

A *configuration* of the machine, then, consists of the contents of the tape, the position of the head, and the current state of the machine (one of its finite set of states). We write configurations like this:

\[
\ldots B B B B a B B \ldots
\]

\[s_0\]

In this configuration the tape contains the word *cat* (with all other squares blank), and the machine is in state \( s_0 \) with the head pointing at the *c*. For clarity, we use the symbol B to denote a blank square.

The operation of the machine is governed by a finite set of *rules* which tell it how to move from one configuration to another. Based on

- the current state; and
- the content of the tape square at which the head is pointing

the rules tell the machine

- which state to go to next;
- what to write on the tape (in the square at which the head is pointing); and
- what to do with the head (move it left, move it right or pause where it is).

We write rules as 5-tuples

\[
\langle q, a, q', a', X \rangle
\]

where \( q, q' \) are states of the machine, \( a, a' \) are symbols and the \( X \in \{L, R, P\} \). For example, the rule \( \langle s_0, c, s_4, b, R \rangle \) means \(^2\)if the machine is in state \( s_0 \) with the head looking at \( c \), then it will move to state \( s_4 \), replace the \( c \) on the tape with \( b \) and move the head one square along the tape to the right. Notice that this rule can be applied in our example configuration above; applying it yields the new configuration:

\[
\ldots B B B B a B B \ldots
\]

\[s_4\]

\(^1\)It could also be carried out by a person (see the introduction to the course) but in this case the person would be basically functioning as a computer. The important point is that the instructions can be followed *routinely* and *mechanically* without any use of additional faculties such as imagination, or intuition, or prior knowledge.

\(^2\)It could also be carried out by a person (see the introduction to the course) but in this case the person would be basically functioning as a computer. The important point is that the instructions can be followed *routinely* and *mechanically* without any use of additional faculties such as imagination, or intuition, or prior knowledge.
We have now described how the machine operates, but not how it starts or stops! The machine has a distinguished start state (say $s_0$) and a distinguished accept state (say $s_1$). The machine starts in the start state with an input word written on the tape and with the head pointing at the first (leftmost) symbol; our first example of a configuration above indicates how the machine would start if the input word were cat. It operates according to the rules until it reaches a configuration in which no rules are applicable. Then it stops.

Each input word $w$ gives rise to a (finite or infinite) sequence of configurations. This sequence is called the computation of the machine on the input $w$.

### 1.4 A Simple Turing Machine.

The definition of a Turing machine is a bit daunting, but the fundamental idea is quite simple. The best way to get a feel for it is with some examples. Let $M$ be the TM with start state $s_0$, accept state $s_1$ and rules

1. $(s_0, a, s_2, B, R)$ pictorially $a$ \\
   $s_0 \rightarrow B\underline{s_2}$

2. $(s_2, a, s_3, B, R)$ pictorially $a$ \\
   $s_2 \rightarrow B\underline{s_3}$

3. $(s_2, B, s_1, a, P)$ pictorially $B$ \\
   $s_2 \rightarrow a\underline{s_1}$

4. $(s_3, B, s_3, a, L)$ pictorially $B$ \\
   $s_3 \rightarrow a\underline{s_3}$

Here is the computation (that is, sequence of configurations) of $M$ on the input word $a$:

```
...B B a B B...
   $s_0$

...B B B B B... by rule 1
   $s_2$

...B B a B B... by rule 3
   $s_1$

M halts in the accept state $s_1$
```

And here is the computation on input $aaa$:

```
...B B B a a a B B...
   $s_0$

...B B B a a a B B... by rule 1
   $s_2$

...B B B B B a B B... by rule 2
   $s_3$

No rule now applies, $M$ halts in a non-accept state
```

---

2 Some books call the start state the initial state and the accept state the terminal or final state.
Finally, on input $aa$ we have:

\[
\begin{align*}
\ldots & B B B a a B B \ldots \\
& s_0 \\
\ldots & B B B B a B B \ldots \text{ by rule 1} \\
& s_2 \\
\ldots & B B B B B B B \ldots \text{ by rule 2} \\
& s_3 \\
\ldots & B B B B B B a B \ldots \text{ by rule 4} \\
& s_3 \\
\ldots & B B B B B a a B \ldots \text{ by rule 4} \\
& s_3 \\
\ldots & B B B B a a a B \ldots \text{ by rule 4} \\
& s_3 \\
\ldots & \ldots \ldots 
\end{align*}
\]

$M$ runs forever without halting!

**Exercise.** Go back and reread the definition of a Turing machine (Section 1.3). It should make a bit more sense now!

**Remark.** For now, we want the rules to specify unambiguously what the machine should do in each situation. To ensure this we insist that for each state $q$ and symbol $a$ there is *at most one rule* of $M$ starting $\langle q, a, \ldots \rangle$, and also that no rule starts with $\langle s_1, \ldots \rangle$ where $s_1$ is the accept state. This property is called *determinism*; we shall see later what happens without it.

**Remark.** To formally specify a Turing machine, one needs to give the alphabet, the set of states, the set of rules, the start state and the accept state. In practice, we can see what the alphabet and states are from the symbols and states which appear in the rules, so it suffices to give only the rules, the start state and the accept state.

**Remark.** As mathematicians, you may think our definition of a Turing machine is a little too “informal”. If so, see the appendix at the end of this chapter.

### 1.5 Acceptance and Languages

**Remark.** We saw in the first example (Section 1.4) that a Turing machine may sometimes carry on computing forever. Clearly a machine which might never stop is not very helpful for solving a problem, so it is useful to have a term for machines which always stop.

**Definition.** A Turing machine is called *halting* if its computation on *every* input word eventually stops.

So what does it mean for a Turing machine to solve a computational problem? For now, we restrict ourselves to solving membership problems for languages. Recall that a Turing machine has a specified *accept state*; this is where the accept state comes into play.

**Definition.** Given a word $\sigma \in \mathcal{A}^*$, we say a TM $M$ *accepts* $\sigma$ if the computation of $M$ on input $\sigma$ halts (eventually) in the accept state.

**Definition.** Given a language $L \subseteq \mathcal{A}^*$, we say that a TM $M$ *accepts* $L$ if

(i) $M$ is halting (on every input word); and

(ii) for every word $\sigma \in \mathcal{A}^*$, $M$ accepts $\sigma \iff \sigma \in L$. 
Warning. Notice the two meanings of “accepts” depending on whether it applies to a single word or to a language.

Remark. Accepting a word \( w \) is just the technical term for “outputting YES” when given the input \( w \). Accepting a language \( L \) is the same thing as solving the membership problem for the language \( L \).

### 1.6 Another Example of a Turing Machine

Our next TM will accept the language

\[
\{ \sigma \in \{0,1\}^* \mid \text{11 is not a subword of } \sigma \} \subseteq \{0,1\}^*
\]

(where \( \nu \) is called a subword of \( \sigma \) if for some words \( \tau, \lambda, \sigma = \tau \nu \lambda \)).

\( M \) has rules

1. \( \langle s_0, 0, s_0, 0, R \rangle \)
   
   \[
   \begin{array}{c|c}
   & 0 \\
   \hline
   s_0 & 0 \\
   \hline
   s_0 & s_0 \\
   \end{array}
   \]

2. \( \langle s_0, 1, s_2, 1, R \rangle \)
   
   \[
   \begin{array}{c|c}
   & 1 \\
   \hline
   s_0 & 1 \\
   \hline
   s_2 & s_2 \\
   \end{array}
   \]

3. \( \langle s_0, B, s_1, B, P \rangle \)
   
   \[
   \begin{array}{c|c}
   & B \\
   \hline
   s_0 & B \\
   \hline
   s_1 & s_1 \\
   \end{array}
   \]

4. \( \langle s_2, 0, s_0, 0, R \rangle \)
   
   \[
   \begin{array}{c|c}
   & 0 \\
   \hline
   s_2 & 0 \\
   \hline
   s_0 & s_0 \\
   \end{array}
   \]

5. \( \langle s_2, B, s_1, B, P \rangle \)
   
   \[
   \begin{array}{c|c}
   & B \\
   \hline
   s_2 & B \\
   \hline
   s_1 & s_1 \\
   \end{array}
   \]

The start state of \( M \) is \( s_0 \) and the accept state is \( s_1 \).

**Idea.** \( M \) moves R(ight) in state \( s_0 \) as long as it doesn’t see a 1. If it does see a 1 it remembers it by changing to state \( s_2 \) and goes R. If it sees another 1 no rule applies and it halts without accepting. If it sees a 0 then the “slate is wiped clean” and it reverts to state \( s_0 \). If it sees a blank in either state \( s_0 \) or state \( s_2 \) then it has reached the end of the input \( \sigma \) without ever encountering successive 1’s and in that case it stands still and accepts.

**Exercise.** Work out the computation of the Turing machine on the input word 001011010. Try more examples until you are confident you understand what is going on.

**Exam/Coursework Guidance.** When asked for a Turing machine, you should always give (at least) the rules, the start and accept states, and an explanation of the “idea” behind the rules.

### 1.7 Palindromes

Next we shall see a TM \( M \) to accept the language of palindromes (words which read the same backwards and forwards). For simplicity, we consider palindromes over the alphabet \( \{0,1\} \), but the same idea works for any alphabet.

**Idea.** On input \( \sigma \) \( M \) remembers the leftmost symbol of \( \sigma \), deletes it (i.e. replaces it by B) and moves off to look for the rightmost symbol. If there isn’t one (because all the word on the tape has now been deleted) \( M \) accepts. If there is such a symbol but it’s not the same as the one \( M \) remembered, \( M \) stops without accepting. Finally, if it is the same, \( M \) deletes it and moves left looking for the leftmost symbol of what remains. If \( M \) can’t find it (again because everything has been deleted) \( M \) accepts. If \( M \) can find it \( M \) starts all over again from there in state \( s_0 \).
Details. The start state of \( M \) is \( s_0 \) and the accept state is \( s_1 \). The rules are:

- **remember and delete**
  \[
  \begin{align*}
  0 & \rightarrow B & r_0 & \quad 1 & \rightarrow B & r_1 \\
  s_0 & \rightarrow B & r_0 & \\
  s_0 & \rightarrow B & r_1 \\
  r_0 & \rightarrow B & c_0 & \quad r_1 & \rightarrow B & c_1 \\
  B & \rightarrow B & s_0 \\
  c_0 & \rightarrow B & s_1 & \quad c_1 & \rightarrow B & s_1 \\
  \end{align*}
  \]

- **find the right end**
  \[
  \begin{align*}
  0 & \rightarrow 0 & r_0 & \quad 0 & \rightarrow 0 & r_1 \\
  r_0 & \rightarrow 0 & r_0 & \quad r_1 & \rightarrow 0 & r_1 \\
  1 & \rightarrow 1 & r_0 & \quad 1 & \rightarrow 1 & r_1 \\
  r_0 & \rightarrow 0 & r_0 & \quad r_1 & \rightarrow 0 & r_1 \\
  B & \rightarrow B & c_0 & \quad B & \rightarrow B & c_1 \\
  \end{align*}
  \]

- **check right end and delete if OK**
  \[
  \begin{align*}
  0 & \rightarrow h & B & \quad 1 & \rightarrow h & B \\
  c_0 & \rightarrow h & B & \quad c_1 & \rightarrow h & B \\
  \end{align*}
  \]

- **find left end and start anew**
  \[
  \begin{align*}
  0 & \rightarrow h & 0 & \quad 1 & \rightarrow h & 1 \\
  h & \rightarrow h & 0 & \quad h & \rightarrow h & 1 \\
  B & \rightarrow B & s_0 \\
  \end{align*}
  \]

**Exercise.** Work out the computation of the Turing machine on the words 010, 0101 and 0110. Try more examples until you are confident you understand what is going on.

**Exercise.** How would you modify this machine to accept the language of palindromes over the Roman alphabet \( \{a, b, \ldots, z\} \)? How many states and rules would the new machine have?

**Exercise.** Go back and reread the definition of a Turing machine (Section 1.3). It should make perfect sense now!

### 1.8 Recursiveness and the Church-Turing Thesis

**Definition.** A language \( L \subseteq A^* \) is called *recursive* or *computable* if it is accepted by some TM. [Remember that for \( L \) to be accepted by a machine \( M \), the latter must halt on all input words!]

**Remark.** The property of being recursive is independent of the overlying alphabet \( A \), so there is no real need to specify the alphabet.

**Example.** In the previous section we saw that there is a halting TM to accept the language of palindromes over the alphabet \( \{0, 1\} \), so this language is recursive.

**Remark.** If \( A \) is a fixed alphabet, then there are only countably many different\(^3\) Turing machines with tape alphabet \( A \). But there are uncountably many languages over \( A \)! Since each TM accepts at most

\(^3\)Here we count two TMs as “the same” if one is obtained from the other just by renaming the states, which does not change the language accepted.
one language (none if it is not a halting TM!), it follows that the enormous majority of languages are not recursive. Nevertheless, you would probably struggle to find an example of a language which is not recursive. We shall see some later.

**Remark.** As defined above, only languages (that is, sets of words) can be recursive. But we can talk about sets of other things being recursive, provided that we have a suitable coding, that is, a way of describing the things using words. For example, a set $S$ of natural numbers is **recursive** if the corresponding language

$$\{ w \in \{0,1\}^* \mid w \text{ is 0 or begins with 1, and represents a number in } S \}$$

is recursive. (We could just as well code them in base 10 over the alphabet $\{0,1,\ldots,9\}$; it doesn’t change which sets are recursive.)

**Remark.** You may have seen other notions of recursiveness or computability, for example when studying Gödel’s theorems. In fact there have been numerous attempts to capture mathematically the idea of an “algorithm” and hence to define what it means for a set to be “computable” or “recursive” but, amazingly, they all turn out to be equivalent. This fact forms the basis of the following claim:

**The Church-Turing Thesis.** Let $L \subseteq A^*$. Then $L$ is recursive if and only if there is an effective process for deciding, given $\sigma \in A^*$, whether or not $\sigma \in L$.

Notice that the term “effective process” has no precise mathematical definition. This means that the Church-Turing Thesis is a philosophical or physical claim, rather than a mathematical one. It does not make sense to ask for a (mathematical) proof of it. However, there is very strong empirical evidence that a Turing machine can do everything that any computer can do, and the Church-Turing Thesis is almost universally believed.

To avoid constantly worrying about the technicalities of Turing machines, it is common practice to describe algorithms relatively informally as “effective processes” and take it as read that a precise Turing machine can be produced to implement them. This is often described as “assuming the Church-Turing thesis”\(^5\). Here is an example of a theorem with two proofs — one is an “effective process” argument while the other is a more formal Turing machine proof.

**Theorem 1.** Suppose $L \subseteq A^*$ is recursive. Then so is $A^* \setminus L$.

**Argument Using the Church-Turing Thesis.** Since $L$ is recursive, there is a Turing machine (call it $M$) to accept $L$. We now describe an effective process. Given a word $w$, we run the machine $M$ with input $w$. If the machine outputs YES then $w \in L$, so $w \notin A^* \setminus L$, and we output NO. If the machine outputs NO then $w \notin L$ so $w \in A^* \setminus L$ and we output YES. We have described an effective process for deciding the language $A^* \setminus L$, so by the Church-Turing Thesis this language is recursive.

**Turing Machine Proof.** Since $L$ is recursive, there is a TM $M$ which accepts $L$. Let $M'$ be the TM which has the same start state as $M$ but a new accept state, $s'_1$, and all the same rules as $M$ together with the rules

$$\langle q, a, s'_1, a, P \rangle$$

whenever no rule in $M$ starts with $\langle q, a, \ldots \rangle$ and $q \neq s_1$ (= accept state of $M$). Then on input $\sigma M'$ runs exactly like $M$ until $M$ halts. If $M$ halts in its accept state then $M'$ halts there too but now doesn’t accept. On the other hand, if $M$ halts without accepting then $M'$ goes on for one more step and accepts. We have shown that $M'$ is halting, and accepts exactly the words which $M$ does not accept, that is, the words in $A^* \setminus L$.

\[\Box\]

Later on we will make increasing use of the first kind of argument, but at present you need some practice manipulating Turing machines.

\(^4\)Also called *Church’s Thesis* or occasionally *Turing’s Thesis*.

\(^5\)Although in my view this is a little misleading. Certainly assuming the Thesis is **sufficient** to justify this approach, but I would argue that it is not **necessary**. This is because the “effective processes” used in such arguments are all composed of steps which are known to be performable on a Turing machine. Thus, even if the Thesis were false, there would still exist Turing machine implementations and the claims would still hold.
Exam/Coursework Guidance. When you are asked to show that a language is recursive, if the question says that “you may assume the Church-Turing Thesis” then either type of argument is acceptable (but the first type is likely to be easier). Otherwise, you should expect to describe a Turing machine, as in the second proof.

1.9 Multitape Turing Machines

Our original definition of a Turing machine had only one tape, but it is often helpful to consider machines with more tapes. For \( k \geq 1 \) a \( k \)-tape TM (\( k \)TM) is just like a one tape TM except that it has \( k \) tapes each equipped with their own head. So the rules for a \( k \)-tape machine look like

\[
\langle q, \bar{a}, q', \bar{a}', \bar{X} \rangle
\]

where

\[
\bar{a} = < a_1, a_2, \ldots, a_k >, \quad \bar{a}' = < a_1', a_2', \ldots, a_k' >, \quad \bar{X} = < X_1, X_2, \ldots, X_k >,
\]

the \( a_i, a_i' \) are tape symbols, the \( X_i \in \{L, R, P\} \) and the meaning of the rule is “if in state \( q \) the \( i \)th head scans \( a_i \) for \( i = 1, 2, \ldots, k \) then change to state \( q' \), change each \( a_i \) to \( a_i' \) and move the \( i \)th head \( X_i \) for \( i = 1, 2, \ldots, k \)”.

Everything else is defined analogously to the single tape case with the input going in on tape 1. Multitape machines appear on the face of it to be more powerful, and certainly they can do many things more efficiently. For example, here is a 2-tape machine to accept language of palindromes over the alphabet \( \{0,1\} \). The start state is \( s_0 \) and the accept state is \( s_1 \):

- Copy the input onto tape 2
- Return head 1 to the start of the input
- Move head 1 right, head 2 left checking they see the same thing all the time
- If heads successfully reach the end accept

Exercise. Work out the computations of this \( 2TM \) for the inputs “010” and “00110101”. Try more examples until you are confident you understand what is going on.

1.10 Multitape vs 1-Tape Turing Machines

The previous example suggests that multitape Turing machines might be more powerful than 1-tape Turing machines. However, if \( L \) is a language accepted by a (halting) multitape Turing machine \( M \), then it is clear intuitively that there is an effective process to decide membership of \( L \): just follow the rules of \( M \). So the Church-Turing Thesis implies that any language accepted by a halting multitape Turing machine is recursive, and hence can also be accepted by a 1-tape Turing machine. In fact, something slightly stronger is true:

**Theorem 2.** Let \( M \) be a multitape TM and \( \mathcal{A} \) an alphabet. Then there is a 1-tape TM \( N \) such that for \( \sigma \in \mathcal{A}^* \),

(i) \( N \) halts on input \( \sigma \) iff \( M \) halts on input \( \sigma \).

(ii) \( N \) accepts \( \sigma \) iff \( M \) accepts \( \sigma \).
**Definition.** We say that two TMs $M$ and $N$ are *equivalent* on an alphabet $\mathcal{A}$, sometimes written $M \equiv N (\mathcal{A})$, if conditions (i) and (ii) above hold for all $\sigma \in \mathcal{A}^*$.

**Outline Proof.** We give an outline of how the machine $N$ works; describing the rules in full detail would be rather technical and not very illuminating! For simplicity assume that $k = 3$; of course the same argument will work with more tapes.

The idea is to glue together $M$’s $k$ tapes to produce a single tape with $k$ tracks or levels, e.g.

<table>
<thead>
<tr>
<th>tape 1</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>tape 2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>tape 3</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

becomes

<table>
<thead>
<tr>
<th>track 1</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>..........</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>track 2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>..........</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>track 3</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

So each square of $M$’s tape will be treated as if it had three vertical levels. We achieve this by changing the alphabet: for each possible combination of three symbols (one per tape), we introduce a new symbol which will represent them on one tape.

There remains one problem: a 3TM has three heads which can point at different places, while our 1TM will only have one head. To solve this, we add yet more symbols to the tape alphabet, so that we can record where the head is on each tape. For each step of the 3TM $M$, the 1TM $N$ will pass along the tape (or the relevant part of it) looking for the three tape squares corresponding to the heads of the 3TM and performing the appropriate action.

While doing this, $N$ will have to record quite a lot of information in its own state; we shall introduce many new states for $N$ as we go along, but of course there will only be finitely many of them. We will describe how $N$ acts relatively informally — the important thing to note is that each of its actions is *completely and uniquely determined by its state and the symbol the head is pointing at*. This ensures that these actions can actually be implemented as formal rules.

Initially on input $a_1 a_2 \ldots a_n$ $M$ starts off in configuration

\[
\begin{array}{cccccccc}
... & B & a_1 & a_2 & \ldots & a_n & B & ...\\
\uparrow & s_0
\end{array}
\]

\[
\begin{array}{cccccccc}
... & B & B & B & \ldots & B & B & ...\\
\uparrow
\end{array}
\]

\[
\begin{array}{cccccccc}
... & B & B & B & \ldots & B & B & ...\\
\uparrow
\end{array}
\]

whilst of course $N$ starts in the configuration

\[
\begin{array}{cccccccc}
... & B & a_1 & a_2 & \ldots & a_n & B & ...\\
\uparrow & s'_0
\end{array}
\]

At this point $N$ does not have its one tape divided into tracks in the intended fashion so it effects this by sweeping R and then L to arrive at the configuration.
\[
\begin{array}{cccccccc}
\ldots & B & a_1 & a_2 & a_3 & \ldots & a_n & B & \ldots \\
\ldots & B & B & B & \ldots & B & B & \ldots \\
\end{array}
\]

\[
\uparrow
\]

\[
\begin{bmatrix}
  s_0 & a_1 & D & \rightarrow \\
  B & D & \rightarrow \\
  B & D & \rightarrow \\
\end{bmatrix}
\]

which is \(N\)'s simulation of \(M\)'s initial configuration. Here expressions like

\[
\begin{bmatrix}
  s_0 & a_1 & D & \rightarrow \\
  B & D & \rightarrow \\
  B & D & \rightarrow \\
\end{bmatrix}
\]

should be thought of as a single tape symbol. Similarly the expression

\[
\begin{bmatrix}
  s_0 & a_1 & D & \rightarrow \\
  B & D & \rightarrow \\
  B & D & \rightarrow \\
\end{bmatrix}
\]

is to stand for a single state of \(N\). The main reason for using such a notation is to make it clear what the state is being used to "remember", in this case the corresponding state and scanned symbols of \(M\). Notice also that \(M\)'s head positions are indicated on \(N\)'s tape by bars and at the simulating stage \(N\)'s head scans the leftmost square containing a bar.

Now suppose that at some stage \(M\)'s configuration is

\[
\begin{bmatrix}
  \ldots & b_0 & b_1 & b_2 & b_3 & b_4 & \ldots \\
\end{bmatrix}
\]

\[
\uparrow q
\]

\[
\begin{bmatrix}
  \ldots & c_0 & c_1 & c_2 & c_3 & c_4 & \ldots \\
\end{bmatrix}
\]

\[
\uparrow
\]

\[
\begin{bmatrix}
  \ldots & d_0 & d_1 & d_2 & d_3 & d_4 & \ldots \\
\end{bmatrix}
\]

and that

\[
\langle q, \begin{bmatrix}
  b_1 \\
  c_3 \\
  d_3
\end{bmatrix}, q', \begin{bmatrix}
  b_1' \\
  c_3' \\
  d_3'
\end{bmatrix}, \begin{bmatrix}
  L \\
  R \\
  P
\end{bmatrix} \rangle
\]

is a rule of \(M\), so \(M\)'s next configuration will be

\[
\begin{bmatrix}
  \ldots & b_0 & b_1' & b_2 & b_3 & b_4 & \ldots \\
\end{bmatrix}
\]

\[
\uparrow q'
\]

\[
\begin{bmatrix}
  \ldots & c_0 & c_1 & c_2 & c_3 & c_4 & \ldots \\
\end{bmatrix}
\]

\[
\uparrow
\]

\[
\begin{bmatrix}
  \ldots & d_0 & d_1 & d_2 & d_3 & d_4 & \ldots \\
\end{bmatrix}
\]

Then corresponding to this stage of \(M\), our 1-tape machine \(N\) is in configuration
Notice that $N$’s state records $M$’s state and scanned symbols, $N$’s head scans the leftmost square containing a bar and the bars indicate $M$’s heads’ positions. The $D$s and $\rightarrow$s in the state will be explained later.

$N$ now simulates this one move of $M$ as follows. First $N$ records what it must do by changing its state to

$$
\begin{bmatrix}
q' & b'_1 & L & \rightarrow \\
c'_3 & R & \rightarrow \\
d'_3 & P & \rightarrow 
\end{bmatrix}
$$

without moving. The $\rightarrow$s here record the position of each of the tape heads of the $M$ (i.e. the barred symbols) relative to the single tape head of $N$.

Notice that this move of $N$ is determined by its state (and scanned symbol, although that is actually irrelevant in this case). Notice also that this state contains all the information required to simulate $M$’s one move. Furthermore there can only be finitely many such states (a necessary requirement for being a TM).

$N$’s head now starts moving R, recording bars seen in the column of arrows and effecting those required tape changes which can be done when moving right ($P$s and $R$s). The configurations are:

$$
\begin{array}{cccccc}
\ldots & b_0 & b_1 & b_2 & b_3 & b_4 & \ldots \\
\ldots & c_0 & c_1 & c_2 & c_3 & c_4 & \ldots \\
\ldots & d_0 & d_1 & d_2 & d_3 & d_4 & \ldots \\
\end{array}
\quad
\begin{array}{cccccc}
\ldots & b_0 & b_1 & b_2 & b_3 & b_4 & \ldots \\
\ldots & c_0 & c_1 & c_2 & c_3 & c_4 & \ldots \\
\ldots & d_0 & d_1 & d_2 & d_3 & d_4 & \ldots \\
\end{array}
\quad
\begin{array}{cccccc}
\ldots & b_0 & b_1 & b_2 & b_3 & b_4 & \ldots \\
\ldots & c_0 & c_1 & c_2 & c_3 & c_4 & \ldots \\
\ldots & d_0 & d_1 & d_2 & d_3 & d_4 & \ldots \\
\end{array}
$$

$$
\begin{array}{cccccc}
\uparrow & q' & b'_1 & L & \rightarrow \\
& c'_3 & R & \rightarrow \\
& d'_3 & P & \rightarrow 
\end{array}
\quad
\begin{array}{cccccc}
\uparrow & q' & b'_1 & L & \leftarrow \\
& c'_3 & R & \rightarrow \\
& d'_3 & P & \rightarrow 
\end{array}
$$

Notice that $N$ knows when it’s seeing the correct ‘$c_3$’ because of the bar. $N$ then changes it to what is required and, in the case of $d_3$, deposits the new bar and records the new scanned (by $M$) symbol in its state. It also changes the “$P$” to a “$D$” (for “done”), to indicate that the operation of this head in this step is now complete.

In the case of $c_3$, $N$ must wait one more move before depositing the bar. $N$ knows however that it must deposit it because the combination “$c_3 \quad R$ $\leftarrow$” tells it this. When it deposits this bar it records the new scanned symbol, $c_4$ in this case, and changes the “$R$” to a “$D$”.

At this point $N$ has all $\leftarrow$s in the last column, indicating that it has seen every bar. It now turns round and heads back to the position of the leftmost bar. As it goes, it makes those tape changes which can only be effected when moving left ($L$s), keeping track still of $M$’s scanned symbols:
The fact that \( L \) are recursive. By the above (i) and (ii) of the theorem.

**Corollary 3.** If \( L_1, L_2 \subseteq A^* \) are recursive then so are \( L_1 \cap L_2 \) and \( L_1 \cup L_2 \).

**Proof.** Let \( M_1, M_2 \) be single tape TM’s accepting \( L_1, L_2 \) respectively. First we show that there exists a 2-tape Turing machine \( M \) accepting \( L_1 \cap L_2 \), as follows. On input \( \sigma \in A^* \), \( M \) copies \( \sigma \) onto tape 2 and returns both heads to the left ends of their copies of \( \sigma \). \( M \) then runs like \( M_1 \) on tape 1. If it halts without accepting so does \( M \). If it halts and accepts then \( M \) runs \( M_2 \) on tape 2 and accepts just if \( M_2 \) does. Clearly \( M \) accepts \( L_1 \cap L_2 \), so by the theorem, \( L_1 \cap L_2 \) is recursive.

The fact that \( L_1 \cup L_2 \) is recursive can be shown similarly. Alternatively, by Theorem 1, \( A^* \setminus L_1 \) and \( A^* \setminus L_2 \) are recursive. By the above \( (A^* \setminus L_1) \cap (A^* \setminus L_2) \) is recursive and so by Theorem 1 again

\[
L_1 \cup L_2 = A^* \setminus [(A^* \setminus L_1) \cap (A^* \setminus L_2)]
\]

is recursive.
1.11 More on Coding

We have already seen how natural numbers can be coded as words over the binary alphabet, and how this allows us to define recursive sets of natural numbers.

**Notation.** If \( n \in \mathbb{N} \) is a natural number we write \( \overline{n} \) for the binary coding of \( n \).

**Exercise.** Show that the language

\[ \{ w \in \{0, 1\}^* \mid w = 0 \text{ or } w \text{ begins with 1} \} \]

of words which code natural numbers is recursive.

From our coding for natural numbers over \( \{0, 1\} \) we can easily develop a coding for \( k \)-tuples of natural numbers over the alphabet \( \{0, 1, @\} \). Each tuple \((n_1, n_2, \ldots, n_k)\) is represented by the word \( \overline{n_1}@\overline{n_2}@\ldots@\overline{n_k} \).

**Proposition 4.** Suppose \( L_1 \subseteq L_2 \subseteq A^* \) and \( L_2 \) is recursive. Suppose there exists a TM \( M \) such that for all \( \sigma \in L_2 \):

(i) \( M \) halts on input \( \sigma \), and

(ii) \( M \) accepts \( \sigma \) \iff \( \sigma \in L_1 \).

Then \( L_1 \) is recursive.

**Proof.** Since \( L_2 \) is recursive there is a halting TM \( M' \) which accepts \( L_2 \subseteq A^* \). So we devise a machine \( N \) which on input \( \sigma \in A^* \) first stores a copy of \( \sigma \) on a separate track 2 and then runs \( M' \) on the original \( \sigma \) on track 1 (just ignoring the stuff on the other track) to check if \( \sigma \in L_2 \). If it isn’t then certainly \( \sigma \notin L_1 \) so this machine halts without accepting. If it is then the machine runs \( M \) on input \( \sigma \) (which is in \( L_2 \)) on track 2 (now ignoring track 1) and accepts just if \( M \) does. (Notice that \( N \) simulates first \( M' \) and then \( M \); this is important because \( M' \) is certain to halt, while \( M \) may not halt on input words which are not in \( L_2 \).)

**Remark.** Suppose we want to show that a given set \( S \) of natural numbers is recursive. Since the set of all codes of natural numbers is recursive, the proposition allows us to assume from the start that our input is already a (code for) a natural number. Similarly for any other system of coding things, provided it has the property that the language of all valid codes is recursive.

1.12 Coding Machines

We have seen how natural numbers can be coded as words, and hence used as input to Turing machines. In this section we shall see how to code mathematical objects of another important type, namely Turing machines themselves.

Let \( M \) be a TM with states \( s_0, s_1, \ldots, s_j \), tape symbols \( \alpha_0, \alpha_1, \ldots, \alpha_k \), where \( \alpha_0 = 0, \alpha_1 = 1, \alpha_2 = \text{B} \), and rules \( r_1, r_2, \ldots, r_p \). We assume for simplicity that \( s_0 \) is the start state and \( s_1 \) is the accept state — this can always be achieved just by renumbering the states. We now “code” everything by words from \( \{0, 1\}^* \) as follows:
Object

The number 0

\[ \hat{0} = 000 \]

[The zero on the left is the number, the zeros on the right are tape symbols.]

The number 1

\[ \hat{1} = 001 \]

The number \( n \) where

\[ n = a_m a_{m-1} \ldots a_0 \text{ in binary} \]

\[ \hat{n} = a_m a_{m-1} \ldots a_0 \]

e.g. \( 13 = 1101, \hat{13} = 001001000001 \)

Notice that we can recover \( n \) from \( \hat{n} \)

\[ s_i, \ 0 \leq i \leq j \]

\[ \bar{s}_i = 010\hat{i} \]

\[ \alpha_i, \ 0 \leq i \leq k \]

\[ \bar{\alpha}_i = 011\hat{i} \]

so \( \bar{0} = 011\hat{0} = 011000 \)

\( \bar{T} = 011\hat{1} = 011001 \)

\( \bar{B} = 011\hat{2} = 011001000 \)

\( R \)

\( \bar{R} = 100 \)

\( L \)

\( \bar{L} = 101 \)

\( P \)

\( \bar{P} = 110 \)

Rule \( r = (s_i, \alpha_i, s_j, \alpha_j, X) \)

\[ \bar{r} = \bar{s}_i \alpha_i \bar{s}_j \alpha_j X \]

Notice that all codes have length divisible by 3 and that we can always recover objects from their codes.

In particular we can recover rule \( r \) from \( \bar{r} \), by looking at successive (i.e. non-overlapping) subwords of \( \bar{r} \) of length 3 and marking off those of the forms 010, 011, 100, 101, 110. This gives

\[ \bar{r} = 010\hat{i}_1011\hat{i}_2010\hat{i}_3011\hat{i}_4 \begin{cases} 100 & \text{if } X = R \\ 101 & \text{if } X = L \\ 110 & \text{if } X = P \end{cases} \]

from which \( i_1, i_2, i_3, i_4 \) can be recovered.

Finally code the TM \( M \) by

\[ \overline{M} = \bar{r}_111 \bar{r}_2111 \ldots 111 \bar{r}_p \in \{0, 1\}^* \]

Notice that we can again recover \( M \) from \( \overline{M} \) by again looking at successive subwords and marking off the 111’s to give \( \bar{r}_1, \bar{r}_2, \ldots, \bar{r}_p \) and so on.

Example. Let \( M \) be the TM accepting the language

\[ \{ \sigma \in \{0, 1\}^* \mid \sigma = \epsilon \text{ or } \sigma \text{ starts with } 01 \} \subseteq \{0, 1\}^* \]

with start state \( s_0 \), accept state \( s_1 \) and rules

\[ r_1 = (s_0, 0, s_2, 0, R) \]

\[ 0 \]

\[ s_0 \rightarrow \begin{array}{c} 0 \\ s_2 \end{array} \]

\[ r_2 = (s_2, 1, s_1, 1, P) \]

\[ 1 \]

\[ s_2 \rightarrow \begin{array}{c} 1 \\ s_1 \end{array} \]

\[ r_3 = (s_0, B, s_1, B, P) \]

\[ B \]

\[ s_0 \rightarrow \begin{array}{c} B \\ s_1 \end{array} \]

Then

\[ \bar{s}_0 = 010\hat{0} = 010000 \]

\[ \bar{s}_1 = 010\hat{1} = 010001 \]

\[ \bar{r}_1 = 010\hat{2} = 010001000 \]

\[ \bar{T} = 011\hat{1} = 011001 \]

\[ \bar{B} = 011\hat{2} = 011001000 \]
Lemma 5. \( \{ \sigma \in \{0,1\}^* \mid \sigma \text{ codes a TM} \} \) is recursive.

Proof. We describe the operation of a 2 tape TM (2TM) \( M \) to accept this language. On input \( \sigma \in \{0,1\}^* \)
\( M \) acts as follows:

(i) If \( \sigma = \epsilon \) halt and accept. [No rules is certainly fine.]

(ii) Otherwise, move head 1 over \( \sigma \) and check that it starts off looking like

\[
010 \hat{i}_1 011 \hat{i}_2 010 \hat{i}_3 011 \hat{i}_4 \gamma
\]

for some \( i_1, i_2, i_3, i_4 \in \mathbb{N} \) and \( i_1 \neq 1 \) and \( X \in \{L, R, P\} \) and \( \gamma = \epsilon \) or \( \gamma = 111 \tau \) for some \( \tau \). If not, reject. [The condition \( i_1 \neq 1 \) is because no rules are allowed to apply in the the accept state \( s_1 \).]

(iii) If \( \gamma = \epsilon \) then accept. Otherwise copy \( 111010 \hat{i}_1 011 \hat{i}_2 \) onto tape 2 and, by repeatedly moving heads 1 and 2 in sync. check that \( \gamma \) does not have the form

\[
\nu 111010 \hat{i}_1 011 \hat{i}_2 \lambda
\]

for some word \( \nu \) of length divisible by 3. If it \( \nu \) does have this form then reject. [In this case there are two rules starting with \( \langle s_{i_1}, \alpha_{i_2}, \ldots \rangle \), which is not allowed.]

(iv) Delete the initial part

\[
010 \hat{i}_1 011 \hat{i}_2 010 \hat{i}_3 011 \hat{i}_4 \gamma 111
\]

of \( \sigma \) on tape 1, erase everything on tape 2 and go back to (i) (now of course with \( \tau \) in place of \( \sigma \)).

It should be clear that all this can be achieved with a TM and that the only way that \( M \) can succeed in accepting a word \( \sigma \) is if \( \sigma \) codes a TM.

\[ \square \]

1.13 Non-Recursive Languages and Unsolvable Problems

We remarked above that many languages are not recursive, but we have yet to see an example of a language which is not recursive. Now that we have the right tools, it is startlingly easy to produce one!

Theorem 6. The language

\[
L_{na} = \{ \sigma \in \{0,1\}^* \mid \sigma \text{ codes a 1TM } M \text{ and } M \text{ does not accept } \sigma \} \subseteq \{0,1\}^*
\]

is not recursive.

Proof. Suppose for a contradiction that it was, and let \( N \) be a TM accepting \( L_{na} \). Then \( \overline{N} \) certainly codes a TM, namely \( N \) so, by definition of \( L_{na} \),

\[
\overline{N} \in L_{na} \iff N \text{ does not accept } \overline{N} \iff \overline{N} \notin L_{na}
\]

since \( L_{na} \) is the language \( N \) accepts. Contradiction! \[ \square \]
Corollary 7. The language \[ L_a = \{ \sigma \in \{0,1\}^* \mid \sigma \text{ codes a TM } M \text{ and } M \text{ accepts } \sigma \} \subseteq \{0,1\}^* \]

is not recursive.

Proof. Clearly \[ L_{na} = \{ \sigma \in \{0,1\}^* \mid \sigma \text{ codes a TM } \} \cap (\{0,1\}^* \setminus L_a). \]

We have seen that the recursive languages are closed under complement (Theorem 1) and intersection (Corollary 3). Thus, if \( L_a \) was recursive then \( L_{na} \) would be too. \( \square \)

Remark. In terms of problems, this result says that the problem of deciding if \( \sigma \in \{0,1\}^* \) codes a TM which accepts this code \( \sigma \) is unsolvable or undecidable. Equivalently (by the Church-Turing Thesis) there is no effective process for deciding, given, \( \sigma \in \{0,1\}^* \), whether or not \( \sigma \) codes a TM which accepts \( \sigma \). Since by Lemma 5 we can decide if \( \sigma \) codes a TM the unsolvable part is to decide, given a code for a TM, whether or not that TM halts and accepts when run on its own code.

Of course for a particular TM \( M \) we may certainly be able to decide this. However there can be no effective way to decide this which works in general for all TMs. There are many similar problems, for example whether or not a TM ever halts when run on its own code (the so called Halting Problem); see Exercise Sheet 1.

Aside for Students of Set Theory. You have probably spotted a resemblance between the method used to produce a non-recursive language and Russell’s Paradox. However, there is nothing paradoxical about non-recursive languages! The difference is that here we consider two different systems of definition (that of sets, which we used to define \( L_{na} \), and that of Turing machines, which we decided could not accept \( L_{na} \)), and argue to the (non-contradictory) conclusion that one is strictly more expressive than the other. In Russell’s case both roles are played by one system of definition (naive set theory) and so one reaches the (contradictory) conclusion that this system is strictly more expressive than itself.

1.14 A Universal Turing Machine

By now you have had lots of practice at the following process: given a 1-tape deterministic halting Turing machine \( M \) (say over the alphabet \( \{0,1\} \)) and a word \( \sigma \), work out the computation of \( M \) on \( \Sigma \), and hence decide whether or not \( M \) accepts \( \sigma \). You have probably convinced yourself that the process for doing this is an entirely mechanical one. So one might ask whether this process can itself be performed by a Turing machine. In other words, is there a Turing machine which can simulate any other Turing machine?

We construct a 3TM \( U \) as follows. The alphabet will be \( \{0,1\} \). The machine is designed to operate with input of the form \( \tau 1111111 \nu \)

where \( \tau \) codes a 1-tape Turing machine (\( M_\tau \) say), and

\[ \nu = a_1 a_2 \ldots a_r \]

is some input for this Turing machine. [We assume for simplicity that the machine \( M_\tau \) itself expects input over the alphabet \( \{0,1\} \), but we shall allow it to use more tape symbols in its internal working; this issue is not very important.]

On input \( \sigma \in \{0,1\}^* \), \( U \) first works out where the divide between \( \tau \) and \( \nu \) is by checking successive blocks of 3 tape symbols in \( \sigma \) until it finds the first pair of adjacent 111’s. (If this or any of the later checks fail \( U \) will just stop without accepting).

Having checked that \( \tau \) codes a Turing machine, \( U \) then copies \( \tau \) onto tape 3 and \( \nu \) onto tape 2. It then replaces \( \sigma \) on tape 1 by

\[ \tau \overline{a}^{6+|\tau|} \overline{a}^{6+|\tau|} \ldots \overline{a}^{6+|\tau|} \]

so the \( \overline{a}^{6+|\tau|} \) all have the same length \( 12+|\tau| = h \) say. [The reason for switching to codes for the symbols is that we want to simulate the operation of \( M_\tau \) on its tape, and this may involve more tape symbols than
has in its tape alphabet. The “padding” with the symbol @ ensures that there will be enough space to replace the coding of \( \overline{\tau} \) with the coding of any other tape symbol used by \( M_r \), without having to shuffle everything else along the tape to make room.]

Next \( U \) replaces \( \nu \) on tape 2 by \( \overline{\sigma} \) and positions head 1 at the left end of \( \overline{\tau} \). At this stage the tapes 1 and 2 contain \( \overline{\sigma_1}@^{h+|\tau|} \overline{\sigma_2}@^{h+|\tau|} \ldots \overline{\sigma_r}@^{h+|\tau|} \) and \( \overline{\sigma_0} \), and the head on tape 1 points at the beginning of \( \overline{\tau} \). We can think of this as simulating the initial configuration of \( M_r \) on input \( \nu \).

\( U \) now scans along \( \tau \) checking for (a code for) a rule starting with \( \langle s_0, a_1, \ldots \rangle \). If successful, say the rule is \( \langle s_0, a_1, s_g, b, L \rangle \), then \( U \) replaces \( \overline{s_0} \) on tape 2 by \( \overline{g} \), the block \( \overline{\sigma_1}@^{h+|\tau|} \) on tape 1 by \( \overline{\sigma_1}@^{h-|\tilde{B}|} \) (notice that this is the same length, and \( h - |\tilde{B}| > 0 \)) and, in this case because the head instruction is "go left", replaces the block of \( h \) B’s to the left of the home square by \( \overline{\sigma_1}@^{h-9} \), finally positioning head 1 under the left end of this \( \overline{B} \). So \( U \)'s tapes 1 and 2 now look like

\[
\overline{B}@^{h-9} \overline{\sigma_1}@^{h-9} \overline{\sigma_2}@^{h+|\tau|} \ldots \overline{\sigma_r}@^{h+|\tau|}
\]

and \( \overline{s_g} \), correctly simulating \( M_r \)'s next configuration:

\[
\ldots \overline{B} \overline{B} \overline{B} b a_1 a_2 \ldots a_r \overline{B} \overline{B} \ldots
\]

\( U \) now continues to simulate \( M_r \) on input \( \nu \) in this way (it should now be clear to you that we can arrange this). Finally, if simulation halts (that is, if \( U \) finds that no rules of \( M_r \) are applicable), \( U \) checks whether tape 2 contains \( \overline{\tau} \). If so it accepts; otherwise, it rejects.

Clearly for any Turing machine \( M \) and word \( \nu \in \{0,1\} \) we have

\[
M \text{ halts on } \nu \iff U \text{ halts on } \overline{M}111111\nu,
\]

\[
M \text{ accepts on } \nu \iff U \text{ accepts } \overline{M}111111\nu.
\]

A TM with this property is called a universal Turing machine. Note that while the machine \( U \) we constructed has 3 tapes, it follows by Theorem 2 that there also exists a 1-tape universal Turing machine. Such machines are mathematical models of the modern day general-purpose programmable computer, with \( \overline{M} \) playing the role of the program.

Corollary. \( \{ \sigma \in \{0,1\}^* \mid U \text{ halts on } \sigma \} \subseteq \{0,1\}^* \) is not recursive.

Proof. Suppose on the contrary this language was recursive. We show that this implies that

\[
L_a = \{ \sigma \in \{0,1\}^* \mid \sigma \text{ codes a TM } M \text{ and } M \text{ accepts } \sigma \}
\]

is recursive, contradicting the previous result.

Using the Church-Turing Thesis it is enough to describe an effective process to decide, given \( \sigma \in \{0,1\}^* \) whether or not \( \sigma \in L_a \). Such a process is as follows: First test if \( \sigma \) codes a TM (effective because we already know the language of such codes is recursive). If so, test if \( U \) halts on input \( \sigma111111\sigma \) (effective by assumption) and if yes run \( U \) on this input, noticing that \( U \) will halt. If \( U \) accepts then \( \sigma \in L_a \). In all other cases \( \sigma \notin L_a \).

Remark. One might imagine that the number of rules required for a Universal TM would be rather large. In fact at one time there was a fashion for producing Universal TM’s (although not necessarily using the same coding for TM’s) and such machines were produced with fewer than 25 rules.

1.15 Recursive Functions

So far we have confined our attention to decision problems but Turing machines can also be used to solve more general computational problems.

Definition. Let \( A_1, A_2 \) be alphabets. We say that a function \( f : A_1^* \rightarrow A_2^* \) is recursive or computable, if there is a TM \( M \) which on input \( \sigma \in A_1^* \) halts in the configuration
where \( f(\sigma) = b_1 b_2 b_3 \ldots b_r \) and \( s_1 \) is the accept state. We say that \( f(\sigma) \) is the output of \( M \) on input \( \sigma \).

Since we have codings of natural numbers, and of tuples of natural numbers, we can also define what it means for functions on the natural numbers to be computable.

**Definition.** We say that a function \( f : \mathbb{N}^k \to \mathbb{N} \) is recursive or computable if there is a TM \( M \) which on input \( n_1 \oplus n_2 \oplus n_3 \oplus \ldots \oplus n_k \) where each \( n_i \in \mathbb{N} \) halts in the configuration

\[
\begin{array}{cccccc}
B & B & b_1 & b_2 & b_3 & \ldots & b_r & B & B \\
& & s_1 & & & & & & \\
\end{array}
\]

where \( b_1 b_2 b_3 \ldots b_r = f(n_1, \ldots, n_k) \) and \( s_1 \) is the accept state.

**Remark.** We don’t really care what happens on inputs not of the form \( a_1 \oplus a_2 \oplus a_3 \oplus \ldots \oplus a_k \), where \( a_1, a_2, \ldots, a_k \) are binary codes for natural numbers. Since we can recognize when an input is of this form, we could always check that first and if it isn’t do whatever we want (for example, halt in a non-accept state).

**Remark.** Similarly, we can define what it means for a function \( f : \mathbb{N}^k \to \mathbb{N}^r \) to be recursive.

**Remark.** Just as for recursive languages, it doesn’t matter if we take \( M \) to be a 1-tape or a multitape Turing machine. In the latter case we assume the output appears on tape 1.

**Theorem 8.** Let \( \mathcal{A}_1, \mathcal{A}_2 \) be alphabets, \( \oplus \) some letter not in either of these, and \( f : \mathcal{A}_1^* \to \mathcal{A}_2^* \). Then \( f \) is recursive iff the language

\[
\{ \sigma \oplus \tau \mid \sigma \in \mathcal{A}_1^*, \tau \in \mathcal{A}_2^*, f(\sigma) = \tau \} \subseteq (\mathcal{A}_1 \cup \mathcal{A}_2 \cup \{ \oplus \})^*
\]

is recursive.

Similarly \( f : \mathbb{N}^k \to \mathbb{N} \) is recursive iff

\[
\{ m_1 \oplus m_2 \oplus \ldots \oplus m_k \oplus m \in \{ 0, 1, \oplus \}^* \mid m = f(n_1, n_2, \ldots, n_k) \}
\]

is recursive (etc.)

**Proof.** See Exercise Sheet 1. \( \square \)

**Remark.** In view of this theorem, the Church-Turing Thesis extends to say that a function \( f : \mathcal{A}_1^* \to \mathcal{A}_2^* \) is recursive if and only if we have an effective process for producing \( f(\sigma) \) given \( \sigma \in \mathcal{A}_1^* \).

**Remark.** This theorem partly explains why languages and decision problems play such an important role in computability and complexity theory. It says that the question of whether a given function is recursive (that is, whether a general computational problem is solvable) is always equivalent to the question of whether a corresponding language is recursive (that is, whether the corresponding decision problem is solvable). So if we fully understood recursive languages, it would only be a short step to fully understand computable functions.

**Example.** The successor function \( s : \mathbb{N} \to \mathbb{N}, n \mapsto n + 1 \) is recursive. The idea for a TM \( M \) to compute \( s \) is as follows:

1. On (binary) input \( a_na_{n-1} \ldots a_1a_0 \in \mathbb{N} \) \( M \) moves right to the first \( B \) and then goes back left one square.
2. \( M \) next moves left changing 1’s to 0’s until it sees a 0 or a \( B \).
3. If it sees a 0 it replaces it by a 1, carries on left (without changing anything more on the tape) until it finds a \( B \), at which point it returns right for one square and goes into the accept state \( s_1 \).
4. If instead it sees a B it replaces it with 1 and goes directly into state $s_1$.

**Exercise.** Write down an exact set of rules for a machine to compute the successor function on binary codings of natural numbers.

In fact all the "everyday" mathematical functions on the natural numbers (for example addition, multiplication, exponentiation, factorial) are recursive. See Exercise Sheet 1 for a chance to convince yourself that addition is recursive.

### 1.16 Non-Deterministic Turing Machines

So far we have insisted that our Turing machines be deterministic; this was enforced by only allowing at most one rule to start $\langle q, a, \ldots \rangle$ for any given state $q$ and symbol $a$, and no rule to start $\langle s_1, \ldots \rangle$ where $s_1$ is the accept state.

A non-deterministic Turing machine (NDTM) is defined just like a (deterministic) Turing Machine, except that we drop the requirement that there be at most one rule starting $\langle q, a, \ldots \rangle$. [For simplicity we keep the requirement that no rule applies in the accept state $s_1$, but this alone does not make much difference.]

Again a computation of an NDTM $N$ on input $a_1a_2\ldots a_n$ is a sequence, possibly infinite, of configurations starting with

$$\ldots \ B \ B \ a_1 \ a_2 \ \ldots \ a_n \ B \ B \ \ldots$$

such that each follows from the previous one by one of the rules. The difference now is that there may be several rules which apply in one configuration, so the computation is *not uniquely determined by the input*. Instead, each input gives rise to a set of possible computations.

**Definitions.** An NDTM $N$ accepts an input word $\sigma$ if there is some computation of $N$ on $\sigma$ which halts in the accept state $s_1$.

An NDTM $N$ halts on an input $\sigma$ if there is some $k \in \mathbb{N}$ such that every computation of $N$ on $\sigma$ halts within $k$ steps.\(^6\)

An NDTM $N$ accepts a language $L \subseteq \mathcal{A}^*$ if for all $\sigma \in \mathcal{A}^*$

(i) $N$ halts on input $\sigma$,

(ii) $N$ accepts $\sigma \iff \sigma \in L$.

**Warning.** Read the above definitions very carefully several times, thinking hard about the quantifiers (the words "every" and "some"). The machine accepts $\sigma$ if any computation reaches an accept state, but only halts on $\sigma$ if every computation halts.

**Exercise.** Can an NDTM accept a word $\sigma$ without halting on $\sigma$?

**Remark.** One intuitive way of viewing non-determinism is to imagine that the machine has the ability to make “guesses”, the outcomes of which always maximise the chances of acceptance. This will become clearer when you have seen some examples.

**Remark.** By exactly the same method as for TM’s (Section 1.12), we may code NDTM’s over the alphabet $\{0, 1\}$. Checking that the language of valid codes for NDTM’s is recursive is actually slightly easier, because we don’t need to check whether there are two rules starting with the same state and symbol. We can also construct a universal NDTM much as we did in Section 1.14 — the main difference is that, when looking for a rule to apply, it may “decide” not to apply that rule, and instead go on and look for another one.

\(^6\)Although it is not obvious, this is actually equivalent to there being no infinite computation on $\sigma$; see Exercise Sheet 1.
1.17 Example of a Non-Deterministic Turing Machine

We describe an NDTM $N$ which uses “guesswork” to accept the language

$$\{ \sigma \tau \lambda \in \{0,1, \@\}^* | \sigma, \tau, \lambda \in \{0,1\}^* \} \subseteq \{0,1, \@\}^*.$$

The start state is $s_0$, the accept state is $s_1$ and the rules are

- **Find @**
  
  $\begin{array}{c|c|c}
  \sigma & 0 & 1 \\
  \hline
  s_0 & s_0 & s_0 \\
  \end{array}$

- **‘Guess’ τ**
  
  $\begin{array}{c|c|c}
  \tau & 0 & 1 \\
  \hline
  s_2 & s_2 & s_2 \\
  \end{array}$

- **‘Guess’ λ**
  
  $\begin{array}{c|c|c}
  \lambda & 0 & 1 \\
  \hline
  s_3 & s_3 & s_3 \\
  \end{array}$

- **Delete λ**
  
  $\begin{array}{c|c|c}
  \lambda & B & B \\
  \hline
  s_4 & s_4 & s_4 \\
  \end{array}$

- **Find left end of first σ**
  
  $\begin{array}{c|c|c}
  \sigma & B & 0 \\
  \hline
  s_5 & s_5 & s_5 \\
  \end{array}$

If $N$ has not already stopped without accepting then what it now has on its tape is of the form

$$\ldots BB\sigma \@ \ldots \@ \@ \nu BB \ldots$$

with the head scanning the left end of $\sigma$. As in Exercise Sheet 1 Question 2(v) we can now add in (deterministic) rules for checking if $\sigma = \nu$, and if so accepting.

**Exercise.** Find both an accepting and a non-accepting computation (one which halts in the accept state and one which halts in a non-accept state) for this machine on the input “101@0101010”. Try more words (and computations!) until you understand what is going on.
1.18 Deterministic vs Non-Deterministic Turing Machines

If you have been looking for trends in the course so far, the following theorem may not come as a surprise!

**Theorem 9.** Let $M$ be an NDTM and $A$ an alphabet. Then there is a (deterministic) TM $N$ such that for all $\sigma \in A^*$,

(i) If $M$ halts on $\sigma$ then $N$ halts on $\sigma$.

(ii) $M$ accepts $\sigma$ iff $N$ accepts $\sigma$.

In particular, if $M$ halts on all input and accepts a language $L$ then $N$ is a halting TM accepting $L$.

**Proof.** The idea is that, on an input $\sigma$, $N$ will run through all possible computations of $M$ on $\sigma$. If $M$ halts on $\sigma$ (remember this means that every computation of $M$ on input $\sigma$ halts within a bounded number of steps) then there can only be finitely many computations, and $N$ will halt when it has seen them all. This ensures that (i) holds.

$N$ halts in an accepting state only when it finds a computation of $M$ which enters the accepting state; this ensures that (ii) holds.

$N$ has 5 tapes and, amongst others, tape symbols $s_0, s_1, \ldots, s_m, r_1, r_2, \ldots, r_k$ where $s_0, s_1, \ldots, s_m$ are the states of $M$ and $r_1, r_2, \ldots, r_k$ are the rules of $M$ ($s_0 = \text{start state}, s_1 = \text{accept state}$). Tape 1 always contains a copy of the input (call it $\sigma$ — we need to remember it for stage 2 later). Tape 4 will be used to generate all finite words composed of the $r_i$ in the order

$$r_0, r_1, \ldots, r_k,$$  

$$\ldots r_k r_k, r_0 r_0 r_0, r_0 r_0 r_1, r_0 r_0 r_2, \ldots, r_0 r_0 r_k, r_0 r_1 r_0, r_0 r_1 r_1, \ldots,$$

It should be clear that we can devise a TM which alters one word composed of $r_i$’s to the next word in this ordering. Clearly any finite computation of $M$ on input $\sigma$ corresponds to the sequence of rules used, in order, and hence corresponds to one of these words in this list. Of course some words in this list will not correspond to computations of $M$.

With $r_{i_1} r_{i_2} \ldots r_{i_p}$ on tape 4 $N$ will use tape 2 to simulate an exact copy of $M$’s tape and tape 3 to code $M$’s state for the computation of $M$ on input $\sigma$ which starts by using the rules $r_{i_1}, r_{i_2}, \ldots, r_{i_p}$ in that order if such a computation exists.

Tape 5 will be used to check that $N$ has not yet seen all computations of $M$ on input $\sigma$. Head 5 seeing a tick $\checkmark$ will indicate this.

Initially $N$ moves into the following configuration in state $f$ where $\sigma = a_1 a_2 \ldots a_t$,

$$a_1 a_2 \ldots a_t$$  

$$\uparrow$$  

$$a_1 a_2 \ldots a_t$$  

$$\uparrow$$  

$$s_0$$  

$$\uparrow$$  

$$r_0$$  

$$\uparrow$$  

$$B$$  

$$\uparrow$$

with all other squares blank. Now suppose that at some stage $N$’s configuration is
Stage 1 $N$ starts to simulate on tape 2 $M$’s tape and on tape 3 $M$’s state for that computation (if it exists) of $M$ in which the sequence of rules used starts $r_{i_1}, r_{i_2}, \ldots, r_{i_p}$. So for example if $r_{i_1}$ is say $\langle s_0, a_1, s_e, a'_1, R \rangle$ then $N$’s next configuration is

\[
\begin{array}{c}
  a_1 a_2 \ldots a_t \\
  \uparrow \\
  a_1 a_2 \ldots a_t \\
  \uparrow \\
  s_0 \\
  \uparrow \\
  r_{i_1} r_{i_2} \ldots r_{i_p} \\
  \uparrow \\
  x \\
  \uparrow \\
\end{array}
\]

in state $f$ where $x = B$ or $\checkmark$ (and all other squares are blank).

Stage 2 At this stage there are two cases to consider.
Case 1 If each $\tau_{i_j}$ is $\tau_k$ check if head 5 sees ✓. If not then no computation of $M$ on $\sigma$ lasts $\geq p$ steps. Since $N$ has not yet halted this means that $N$ has already seen/simulated every possible computation of $M$ on $\sigma$ and none of them lead to acceptance. In this case $N$ halts without accepting.

On the other hand, if head 5 does see a ✓ and each $\tau_{i_j}$ is $\tau_k$ $N$ resets its tapes to

\[
\begin{align*}
  a_1 & \quad a_2 & \ldots & \quad a_t \\
  \uparrow & & & \\
  a_1 & \quad a_2 & \ldots & \quad a_t \\
  \uparrow & & & \\
  s_0 & & & \\
  \uparrow & & & \\
  \tau_0 \tau_0 \ldots \tau_0 & \quad (p + 1 \text{ copies of } \tau_0) \\
  \uparrow & & & \\
  B & & & \\
  \uparrow & & & \\
\end{align*}
\]

in state $f$. Notice that in this case this word $\tau_0 \tau_0 \ldots \tau_0$ of length $p + 1$ is the next word in the above ordering after $\tau_{i_1} \tau_{i_2} \ldots \tau_{i_p}$.

Case 2 Finally if case 1 does not apply leave tape 5 as it is and reset tapes 1,2,3,4 in state $f$ to

\[
\begin{align*}
  a_1 & \quad a_2 & \ldots & \quad a_t \\
  \uparrow & & & \\
  a_1 & \quad a_2 & \ldots & \quad a_t \\
  \uparrow & & & \\
  s_0 & & & \\
  \uparrow & & & \\
  \tau_{j_1} \tau_{j_2} \ldots \tau_{j_p} & \quad \tau_{j_1} \tau_{j_2} \ldots \tau_{j_p} \\
  \uparrow & & & \\
  x & & & \\
  \uparrow & & & \\
\end{align*}
\]

where $\tau_{j_1} \tau_{j_2} \ldots \tau_{j_p}$ is the word after $\tau_{i_1} \tau_{i_2} \ldots \tau_{i_p}$ in the above ordering.

Round Up. If $M$ has an accepting computation on $\sigma$ then $N$ will eventually find it and accept. Notice that this will occur before $N$ is required to halt through having seen all possible computations of $M$. On the other hand if there is a number $p$ such that no computation of $M$ on $\sigma$ goes on for more than $p$ steps then there will be a least such $p$ and unless $N$ has already accepted, $N$ will eventually reach words of length $p + 1$ on tape 4 and never replace the $B$ on tape 5 by a ✓. Hence when $N$ has run through all such words tape 5 will still hold a $B$ and $N$ will halt without accepting, just as $M$ must have done.

Appendix A: Formal Definition of a Turing Machine

The material in this appendix is for interest only. It does not form part of the syllabus and will not be examined.

As mathematicians, you may think our definition of a Turing machine (Section 1.3) is rather informal and hand-wavy. In this (non-examinable) appendix we briefly indicate how this definition can be made entirely formal and mathematical. The aims are twofold: firstly to convince you that it is possible to put everything on an entirely rigorous footing, and secondly to illustrate why we haven’t done so!

Formally, a non-deterministic Turing machine (NDTM) is a 5-tuple $(\mathcal{A}, Q, q_0, q_1, R)$ where
• \( \mathcal{A} \) is a finite set (called the *alphabet*) containing the symbol \( \mathbb{B} \) (called the *blank symbol*);
• \( Q \) is a finite set (of *states*);
• \( q_0 \in Q \) is called the *initial state*;
• \( q_1 \in Q \) is called the *accept state*; and
• \( R \subseteq (Q \setminus \{q_1\}) \times \mathcal{A} \times Q \times \mathcal{A} \times \{-1, 0, 1\} \) is a finite set of *rules*.

The machine is called a *(deterministic) Turing machine (TM)* if for every \( q \in Q \) and \( a \in \mathcal{A} \),

\[
|R \cap (\{q\} \times \{a\} \times Q \times \mathcal{A} \times \{-1, 0, 1\})| \leq 1.
\]

A *configuration* of \( M \) consists of a triple \((q, h, f)\) where \( q \in Q \), \( h \in \mathbb{Z} \) and \( f \) is a function from \( \mathbb{Z} \) to \( \mathcal{A} \).

The binary relation \( \rightarrow \) is defined on the set of configurations by \((q, h, f) \rightarrow (q', h', f')\) if and only if \((q, f(h), q', f'(h), h' - h) \in R\), and \( f'(g) = f(g) \) for all \( g \in \mathbb{Z} \setminus \{h\} \).

If \( w = w_1 \ldots w_n \in \mathcal{A}^* \) (where each \( w_i \in \mathcal{A} \)) then we define \( I_w \) to be the configuration \((q_0, 1, f)\) where \( f(i) = w_i \) for \( 1 \leq i \leq n \) and \( f(i) = \mathbb{B} \) for \( i < 1 \) and \( i > n \).

Let \( \rightarrow^* \) be the reflexive, transitive closure of binary relation \( \rightarrow \). We say that \( w \in \mathcal{A}^* \) is *accepted* by \( M \) if \( I_w \rightarrow^* (q_1, h, f) \) for some \( h \) and \( f \).

**Exercise (optional!).** Work out how this approach matches up with the more informal definition in Section 1.3. Now try to prove Theorem 1 from Section 1.8 using this formalism.