Chapter 11

Introduction to Computability and Turing Machines

So far, we have seen some reasonably simple models of computers in the form of deterministic and non-deterministic finite automata, as well as a very elegant notation in the form of regular expressions for a certain class of languages. We have seen that all these models (including the regular expressions notation) are equivalent in terms of the class of languages that they accept/describe. This led us to conclude that this class, the regular languages, must be quite important. But we have also seen that some relatively simple languages, such as $L = \{0^n 1^n | n \geq 0\}$, cannot be described by a regular expression or accepted by any form of the finite automata we considered. This showed that finite automata is quite limited as a model, because if it cannot even answer a reasonably simple question like "is an arbitrary string of 0's and 1's of the form $0^n 1^n$, there is no hope that it will be able to answer some more complicated questions and problems.

Broadly speaking, our aim is to investigate the theoretical limits of computers, if any even exist (and it is not trivial to show that there are, indeed, limits). To do this, we need some better model of a computer, that will more closely resemble how the real computers operate and that will, therefore, be more useful in discovering the limitations of computers. But, even more fundamentally, we need to define precisely what do we mean by a *computation*. Intuitively, a computation is something that can be carried out by a machine. In order for something to be carried out by a machine, it is reasonable to require two things from it:

1. The method should be describable in a finite amount of space (description must be incorporated into the machine somehow).

2. The computation must comprise of discrete steps, each of which can be carried out mechanically.

The second point above is especially important, and we have already encountered it in our study of languages. That carrying out the computation should be *entirely mechanical* means that there must not be any 'creative' work involved in carrying it out. The computation might comprise of a huge (but still not unlimited) number of steps, but each step should be simple enough that it can be carried out mechanically by a machine. What exactly constitutes a valid 'step' of a computation is a key ingredient of any definition of a computation. We already had a glimpse of this in our study of finite automata, where the 'computation' was reading a string and deciding whether to accept it or not, and a step of this computation, in the case of DFAs, was reading one input symbol from a string and moving to a new state. But we know that the modern computers can do much more than this, not least because they have memory. So, this notion of a computing step is too basic and we need a more powerful one.

It seems obvious that we should also require any computation to end after a finite time, but for the reasons that we will see a bit later, this is a bit trickier.

The conditions above describe the notion of an *effective procedure* or an *algorithm*. We all know examples of algorithms, from the simpler ones such as multiplying two integer numbers, to the more complex ones such as sorting algorithms. All these examples can be broken down into sequences of simple basic steps that can be performed automatically, without any creativity. Most of the algorithms also require an input (such as, for example, a pair of integer numbers for the integer multiplication algorithm), and will produce different outputs for different inputs. Nowadays, we tend to think of algorithms in terms of writing computer programs to carry them out, where the computer programs consist of sequences of simple individual instructions. But just as well we might imagine computation being carried out mechanically by a person. In fact, the term ’computer’ originally used to describe a person who did routine calculations, e.g. produce tables for navigation.

An algorithm can also be though of as a sort of a black box, which takes an input and by purely mechanical means produces an output. For example, we can thought of it as a black box that computes some function f, meaning that it receives as an input argument x, and produces as an output the value $f(x)$. If the function can be computed in this way, purely mechanically, we call such function *computable*. It is not immediately clear that there are any functions that are *not* computable. But we will see some very important examples of functions that are not computable.

The first thing that we need to do in our discussion is to introduce a formal model of an algorithm. In fact, we are going to introduce a formal model of a computer and of a computation, including a precise notion of what constitutes a single step of a computation. The algorithm will then be the same thing as a “program” for our computer and the computation will be application of an algorithm to some particular input. This might seem a bit strange, because in our everyday work with programming, we clearly distinct between an algorithm and its concrete implementation in some programming language and on some computer. For example, we have seen Dijkstra’s algorithm for finding the shortest path between two nodes in a graph described in an abstract way (usually using some form of a pseudocode), and this abstract notion was different than a particular function in a particular programming language that implements Dijkstra’s algorithm. Implementation of Dijkstra’s algorithm in Python is quite a different thing that the implementation of the same algorithm in Java. In our consideration here, we do not make this distinction, because the very thing that we want to do is to concretise the abstract notion of an algorithm. Therefore, there will be no difference between an algorithm and a “program” that implements that algorithm in our model.

An important thing to note is that since we are trying to formalise something of which we so far only have a somewhat vague intuition, we cannot say for sure whether our formal model is ’good’. But there are good arguments that the computing models that we are going to consider indeed does capture exactly the notion of what is mechanically computable. The model that we are going to use in the rest of the material is the model of a *Turing Machine*.

11.1 Basic Turing Machines

Turing Machine as a model of a computer was introduced by the British mathematician Alan Turing in 1936. It is universaly accepted as a model of computation, as it manages to capture the computational capabilities of computers, while remaining very simple. Informally, a Turing Machine (or, for short, TM, as we will call it sometimes) consists of a *finite state control*, one-way infinite *tape* divided into cells, and *head*, which scans one tape cell at a time (see Figure 11.1). Finite state control can be in one of a finite number of states. Each cell of a tape holds one of finitely many *tape symbols*. There is a special tape symbol called the *blank* symbol (B in our example). The tape has the leftmost cell (the cell containing a_1 in our example, but is infinite to the right. Initially, the leftmost n cells hold an input, which is a string over some *input alphabet*. The remaining cells of the tape hold the blank symbol.

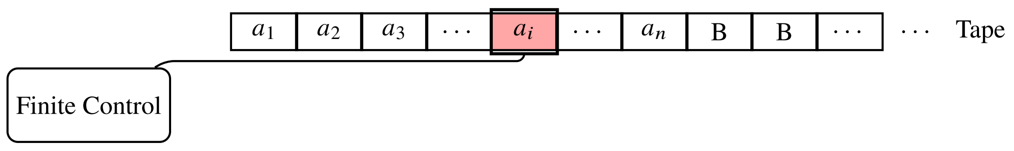
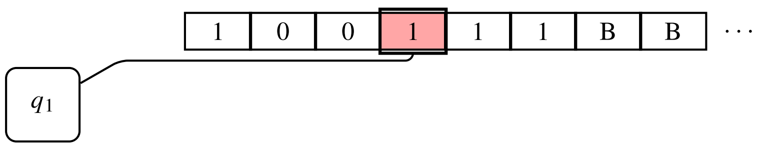


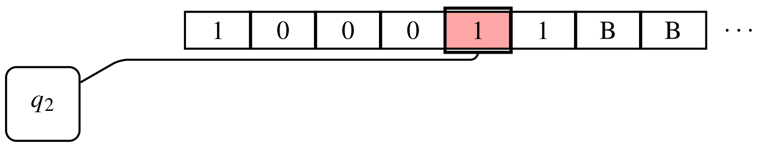
Figure 11.1: Basic Turing Machine

One move (or one step) of the Turing Machine consists of four mini-steps:

- reading the symbol in the cell that is currently scanned;
- changing the state of the Turing Machine;
- overwriting the symbol in that cell with a new symbol;
- moving the tape head one cell to the left or to the right of the currently scanned cell;



(a) A Turing Machine Before the Move



(b) A Turing Machine After the Move

Figure 11.2: One Move of a Turing Machine

An example of this is given in Figure 11.2. The Turing Machine in that example is in the state q_0 and it scans the fourth cell of its tape. Upon reading the symbol 1 in it, it changes its state to q_2, replaces the symbol 0 on the tape with 1 and moves the tape head one cell to the right.

We can note many similarities between the Turing Machine model and a DFA model, but also some important differences. We can see a DFA as a restricted Turing Machine which, at every step, reads a symbol from the tape, changes the state, overwrites the symbol on the tape with that same symbol and moves one cell to the right. The key feature of a Turing Machine that makes it much more powerful than a DFA is not that it can move the tape head also to the left of the currently scanned cell - there exists an extension of a DFA called *two-way* DFA that can move to the left and to the right in the string that it reads, and it can be shown that this extension does not increase the power of the model, as it accepts only regular languages. The key advantage is in being able to write a new symbol in a cell.

11.1.1 Formal Definition of a Turing Machine

Definition 11.1: Turing Machine - Formal Definition

A Turing Machine M is a 7-tuple $(Q, \Sigma, \Gamma, \delta, q_0, B, F)$, where

- Q is a finite set of states;
- Σ is an input alphabet;
- Γ is a set of allowed tape symbols;
- $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$ is a transition function (or *next move* function). However, q may be undefined for some values;
- q_0 is the starting state;
- B is a special, blank symbol from Γ, $B \notin \Sigma$;
- $F \subseteq Q$ is the set of final states;

Note that $\Sigma \subset \Gamma$, as every input symbol may appear on the tape. Input symbols are the symbols that the input strings are composed of. Γ contains, in addition to all the symbols from Σ, also other symbols that can appear on the tape at any point in the operation of the machine. Notice that, compared to Σ, Γ always contains at least one extra symbol, which is the blank symbol B. $\delta(p, A) = (q, B, L)$ means that if the tape head is in the state p and the current cell it scans has the symbol A in it, in the next move M will move to the state q, replace A with B in the currently scanned cell and move one cell to the left. Obviously, if the third component of the tuple that the function returns is R rather than L, the tape head moves one cell to the right.

So, how does M operate on an input string w? We will call the operation of M on some input string a *computation*. It starts in the state q_0, with the tape head scanning the first cell, which is the cell that contains the first symbol of w. Let this symbol be a. The transition function δ for a pair (q_0, a) returns the next move of M. If $\delta(q_0, a) = (p, X, R)$, for some tape symbol X, the tape head moves to the state p, replaces a with X and moves one cell to the right. If w is an empty string, then the symbol in the first cell is going to be B, and the next move would be specified by $\delta(q_0, B)$. The next move starts by scanning the symbol in the cell to which the tape head moved in the previous step. The computation proceeds in this way until one of the terminating conditions are satisfied. The terminating conditions are:

1. The tape head is in the state q, is scanning the cell which contains symbol X, $\delta(q, X)$ is undefined and q is not an accepting state.
2. The tape head enters one of the final states from F.
3. The tape head is in some state q, scans the leftmost cell of the tape, that cell contains symbol X and $\delta(q, X)$ specifies that the tape head moves to the left in the next move.

In the cases 1 and 3, the string w is rejected by M. Only in the case that the terminating condition that was satisfied is 2 will M accept string w. Note that the final states of a Turing Machine are, indeed, final, since the operation of a TM terminates as soon as one of them is entered.

Note, however, that there is also one more possibility for a computation of M on string w. It can happen that there is always a next move and that also M never enters one of the final states. Thus, M never reaches one of the terminating conditions and the computation proceeds indefinitely. This could not have happened in any form of a finite automata that we considered. If M, for some input string w, eventually reaches one of the terminating conditions, then we say that M *halts* on string w. Therefore, there are three possible outcomes of a computation of M on some string w:

- M accepts string w by halting in an accepting state after a finite number of steps.
- M rejects string w by halting in a non-accepting state or "sliding off the tape" (condition 3).
- M does not halt on w.

In the third case, when M does not halt on w, we neither say that M accepts w, nor that M rejects w. Acceptance of a string by a TM is, therefore, not a binary decision.

11.1.2 Instantaneous Description and Formal Definition of a Language Accepted by a TM

In order to talk more formally about acceptenace (or non-acceptance) of a string by a TM, we need to introduce a concept of transition function for strings for a TM, similarly as we did for different flavours of finite automata. There is, however, a problem. The step of a finite automata always lead to reading the next symbol of an input string (or, at worst, reading the same symbol in the case of an ϵ-transition. The only relevant symbols were always the symbols to the right of the currently read symbol of a string. This is not the case with TMs, as their tape head can also move to the left. Therefore, both the symbols left from the currently scanned symbol, and the symbols to the right of the currently scanned one matter. For this reason, instead of extending the transition function of a TM to strings, we define a new concept of an *instantaneous description* of a TM.

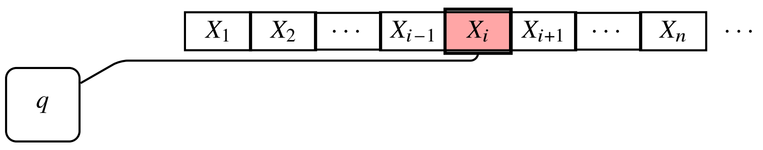
Instantaneous Description (ID) of a Turing Machine M is denoted by $\alpha_1 q \alpha_2$, where

- q is the state in which the TM currently is;
- α_1 is the string of tape symbols that are to the left of the currently scanned cell;
- α_2 is the string of tape symbols from the currently scanned cell to the rightmost non-blank symbol. If there are no non-blank symbols to the right of the currently scanned cell and the currently scanned cell holds the blank symbol, then α_2 is B.

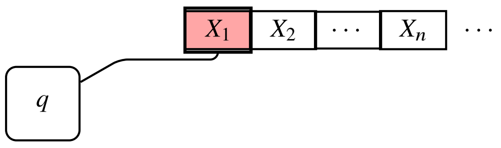
The currently scanned cell holds the first symbol of α_2. Looking back at the example TM in Figure 11.2, the ID of the configuration in Figure 11.2a is $100q_1111$ and the ID of the configuration in Figure 11.2b is $1000q_211$.

Now we formally describe one move of M. Let $X_1X_2 \dots X_{i-1}qX_i \dots X_n$ be an ID (Figure 11.3. We need to consider different possibilities for $\delta(q, X_i)$

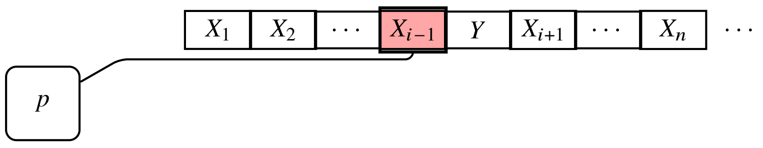
- Suppose that $\delta(q, X_i) = (p, y, L)$.



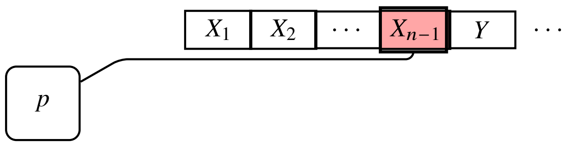
(a) The Original Configuration of M.



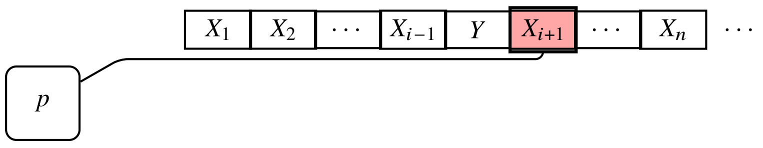
(b) $\delta(q, X_i) = (p, Y, L)$ and $i = 0$, The tape head is scanning the leftmost cell, and would need to move further left. Not allowed.



(c) The Configuration after the move where $\delta(q, X_i) = (p, Y, L)$ if $i > 1$ and $i < n + 1$.



(d) The Configuration after the move where $\delta(q, X_i) = (p, Y, L)$ if $i > 1$ and $i = n + 1$.



(e) The Configuration after the move where $\delta(q, X_i) = (p, Y, R)$.

Figure 11.3: The Next Move of a Turing Machine

- If $i = 1$, there is no next move, since the tape head needs to move to the left of the leftmost tape cell, which is not allowed (Figure 11.3b).
- If $i > 1$ and $i < n+1$, then the ID of the configuration that M moves to is $X_1 X_2 \ldots X_{i-2} p X_{i-1} Y X_{i+1} \ldots X_n$ (Figure 11.3c).
- If $i = n + 1$, then $X_n = B$ and the tape head is scanning the first of the infinite sequence of the blank symbols (Figure 11.3d). The ID of the configuration that M moves to is $X_1 X_2 \ldots q X_{i-1} Y$.

• Suppose now that $\delta(q, X_i) = (p, Y, R)$. The ID of the configuration that M moves to is then $X_1 X_2 \ldots X_{i-1} Y p X_{i+1}$ (Figure 11.3e).

If M moves from the configuration described by ID I_1 to the configuration described by ID I_2 in one move, we denote $I_1 \vdash^M I_2$. Where no confusion can arise, we omit the M and just write $I_1 \vdash I_2$. Thus in the example of Figure 11.3c, we write

$$X_1 X_2 \ldots X_{i-1} q X_i \ldots X_n \vdash X_1 X_2 \ldots X_{i-2} q X_{i-1} Y \ldots X_n.$$

If M moves from the configuration described by ID I_1 to the configuration described by ID I_2 in a finite number of moves (including possibly 0), we write $I_1 \vdash^* I_2$.

With this, we can now formally define the language accepted by a Turing Machine M.

Definition 11.2: Language of a Turing Machine

Let $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$ be a Turing Machine. The language accepted by a Turing Machine (or the language of a Turing Machine) M is defined as

$$L(M) = \{w | q_0 w \vdash^* \alpha_1 p \alpha_2 \text{ for some strings } \alpha_1, \alpha_2 \in \Gamma^* \text{ and some state } p \in F\}.$$

In other words, a string w is accepted by a Turing Machine M if, starting from the start state q_0 and with the string w on its input tape, alligned to the left end of it, M eventually enters one of the accepting states.

11.1.3 Totally Decidable, Partially Decidable and Unrecognisable Languages

Recall that in Finite Automata, the class of languages accepted by DFAs was called the class of *regular languages*. The situation is a bit more complicated with Turing Machines, as there are different 'degrees' to which the language is accepted. The situation is further complicated by the fact that these different classes have different names in many different textbooks, which can make matters confusing.

Definition 11.3: Partially Decidable Languages

Language L over some alphabet Σ is *partially decidable* (also called *recognisable* or *recursively enumerable* in some literature) if there exists a Turing Machine M that accepts L.

It is worth stopping here for a moment. Recall that when we dealt with finite automata and regular expressions, when we were given a language L, as soon as we found some form of finite automata or a regular expression that accepts/describes this language, we were done. Any form of automata that we considered had a nice feature that, when it is given a string w as an input, it would always finish reading w in a finite number of steps. Therefore, the automata would be able to *decide* in a finite amount of steps whether the string w is accepted or rejected. Finding a finite automata M that accepts L would then solve our problem of deciding whether any string w belongs to this language - for any string w, we could simply give it as an input to M and M would decide after a finite number of steps whether w belongs to L or not. The situation is not that simple with Turing Machines. Recall that the language L is accepted by a Turing Machine M if M halts in an accepting state for every string in L. But we don't know what happens for the strings not in L! All we know is that they are not accepted. But we don't know if for all of them M halts in a non-accepting state or it possibly never halts. Both of these mean that the string is not accepted. So, it is perfectly possible for M from Definition 11.3 not to halt at all on some strings from L.

Definition 11.4: Totally Decidable Languages

Language L over some alphabet Σ is *totally decidable* (also called *decidable* or *recursive* in some literature) if there exists a Turing Machine M that accepts L and M halts on all strings from Σ^*.

This looks better! If the language L is totally decidable, then there exists a Turing Machine M that halts on all inputs and accepts only the strings from L. Therefore, such a Turing Machine can be used to decide if a given string w belongs to the language L.

Definition 11.5: Unrecognisable Languages

Language L over some alphabet Σ is *unrecognisable* if there does not exist a Turing Machine M that accepts L.

An equivalent from non-regular languages for finite automata, unrecognisable languages are the languages for which we cannot find a Turing Machine that accepts them.

Two questions can pop into mind when we look at the definitions above. Firstly, do there exist languages that are unrecognisable? Is there a single language L for which it is impossible to find a Turing Machine that accepts it? When we studied regular languages, the similar question appeared, and we were able to answer it pretty easily, with the Pumping Lemma. However, Turing Machines are far more powerful that finite automata, and it is pretty easy to design Turing Machines for many languages we have proved to be non-regular. It turns out that there exist languages that are unrecognisable, but finding an example of an unrecognisable language is far from an easy feat.

Remark 11.1: Unrecognisable Languages Exist

Digging a bit deeper into Maths reveals to us that it is pretty obvious that unrecognisable languages must exist. If we are given as simple alphabet as $\Sigma = \{0, 1\}$, the set of all languages over Σ is uncountably infinite. As we will show later on, the number of Turing Machines over the alphabet $\{0, 1\}$ is countably infinite. There is, therefore, many many more languages over $\{0, 1\}$ than there are Turing Machines over the alphabet $\{0, 1\}$. In fact, if you were to pick a random language over the alphabet $\{0, 1\}$, the probability of picking a partially decidable or totally decidable language would be 0. Similarly as if you were to pick a random number from the interval $[0, 1]$ of real numbers, the probability of picking a rational number would be 0.

The second question we can ask is whether the classes of Partially Decidable and Totally Decidable languages are different. Every totally decidable language is obviously partially decidable too. So, totally decidable languages are a subset of partially decidable. But are they a proper subset? That is, does there exist a language that is partially decidable, but not totally decidable. To find such a language, we would need to find a language for which there exists a Turing Machine that accepts it, but there does not exist a Turing Machine that accepts it and that halts on all inputs. Or, in other words, that every Turing Machine that accepts such language has to run infinitely (without halting) on some input strings. This is even harder than finding an undecidable language. But it is also something we will do in the course of our discussion.

To summarise our discussion, in our study of computability, we are going to prove two remarkable results:

1. There exist languages that are unrecognisable.
2. There exist languages that are partially decidable, but not totally decidable.

11.1.4 An Example of a Turing Machine

Let us now see one example of a Turing Machine. This is one of the rare examples where we will go through the whole process of defining a Turing Machine, including defining the whole transition function. Most other times, an informal (yet precise) description of a Turing Machine will suffice.

Example 11.1: Turing Machine for 0^n1^n

Find a Turing Machine M that accepts the language $L = \{0^n1^n | n \geq 1\}$.

We know that the language L is not regular, so this is one example where a Turing Machine accepts a language that cannot be accepted by any form of a finite automata that we considered.

The principle of the Turing Machine M is quite simple. We mark off a single 0 with X at the start of the input string, then move the tape head over any of the remaining 0's until we find the first 1. Then we mark off the first 1 with Y and move the tape head back to the left until we find the first X. Then we move the head to the right, mark off the next 0 with X and repeat this whole process, moving the tape head to the right over all 0's and Y's and marking off the first unmarked 1. If we find a 0 after the first Y, we reject the string, as it is not of the form $\mathbf{0}^*\mathbf{1}^*$. When there are no more 0's to mark off, we check whether there are remaining unmarked 1's. If there are, the string is rejected (this is the case where there are more 1's than 0's in the input string). Again, if we find another 0 when looking for unmarked 1, it means the string is not of the form $\mathbf{0}^*\mathbf{1}^*$. If, however, when checking whether there are remaining unmarked 1's, we encounter a B, then the string is accepted, as this B marks the end of the input string. If, at some point, we mark off a 0 and then do not find any 1 to the right of it on the tape, it means there are more 0's than 1 in the input string, and we also reject the string.

Our Turing Machine has 5 states. The states are:

- q_0 - the start state. If the symbol scanned in this state is 0, this 0 is replaced by X, the tape head moves to the right and the state q_1 is entered. The state q_0 has another purpose. If Y is encountered in it, it means that all the 0's in the prefix of the string have been replaced by X's. In this case, the state q_3 is entered.

- q_1 - the state in which the tape head moves to the right over any 0's and Y's until it encounters a 1. This 1 is then replaced by Y and the tape head moves to the left and enters the state q_2.
- q_2 - the state in which the tape head moves to the left over any Y's and 0's until it finds the first X. Then moves to the right and enters the state q_0.
- q_3 - the state which is entered when there are no more 0's in the prefix of the input to be replaced by X. We now need to check if all 1's have been replaced by Y. The tape head goes to the right, going over all Y's. If B is found at some point, that means the end of the input has been reached and no 1's found. In this case, the state q_4 is entered.
- q_4 - accepting state. Nothing to be done in it, since M will immediately halt when it enters this state.

The complete transition function of M is given in the following table

δ	0	1	X	Y	B
q_0	(q_1, X, R)	-	-	(q_3, Y, R)	-
q_1	$(q_1, 0, R)$	(q_2, Y, L)	-	(q_1, Y, R)	-
q_2	$(q_2, 0, L)$	-	(q_0, X, R)	(q_2, Y, L)	-
q_3	-	-	-	(q_3, Y, R)	(q_4, B, R)
q_4	-	-	-	-	-

Note that in the case where the symbol that shouldn't be there is scanned, the corresponding transition is nonexistent, so the machine simply halts in a non-accepting state. For example, in the state q_3 when the tape head goes to the right over all Y's, if 0 or 1 is encountered, that means that either the input string is not of the form $\mathbf{0}^*\mathbf{1}^*$ or it is of that form, but there are more 1's that 0's, both of which lead to a rejection of the string. Note also that M halts for all inputs. This means that the language L is totally decidable.

Example of operation of this machine is given in the following video

Video 11.1: Example of Operation of a Turing Machine

Demonstration of operation of the Turing Machine M on input strings 0011 and 00110.

11.2 Modifications of Turing Machines

The Turing Machine model that we presented is a basic model and there are many different modified versions of that model. One of the reasons the model we presented is universally accepted as a general model of a computation is that many of these versions are equivalent to the basic version, in terms of the languages that they accept. Here we will describe two such versions.

11.2.1 Two-Way Infinite Tape Turing Machine

One important modification is a *two-way infinite tape Turing Machine*, where the tape is infinite both to the left and to the right. At the start of the computation on a string w, the string is stored on the tape, and the tape head is scanning the first symbol of the string, the same as for the basic model. However, in the two-way infinite tape model, there is also an infinite sequence of cells holding the blank symbol to the left of the cell holding the first symbol of string w (see Figure 11.4). Compared to the basic Turing Machine, there is no left end of the tape here, so one of the terminating conditions for the basic Turing Machine (the one where the next move takes the tape head to the left of the leftmost cell) is no longer valid here.

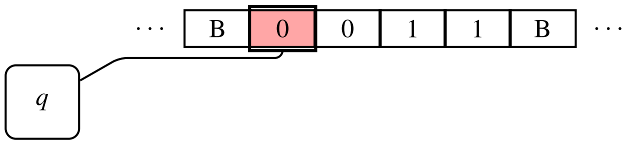
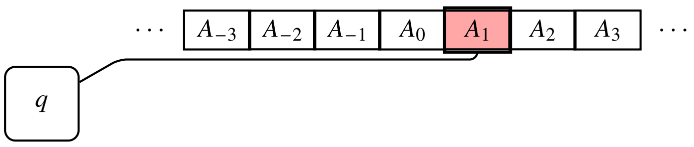


Figure 11.4: A Two-Way Infinite Tape Turing Machine with the String 0011 as an Input.

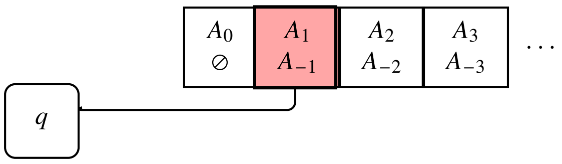
Theorem 11.1: Equivalence Between the Basic and Two-Way Infinite Tape Models

L is accepted by a Turing Machine with a two-way infinite tape if and only if it is recognised by a basic Turing Machine.

Proof Outline. One direction is obvious - if a basic Turing Machine accepts some language, then certainly the two-way infinite tape Turing Machine with the same transitions will also accept that language. The other direction is non-trivial. Let M be a two-way infinite tape Turing Machine, with the transition function δ. We need to find a way to simulate M with a basic Turing Machine M' with its transition function δ'. The key part here is representing the two-way infinite tape of M with a one-way infinite tape of M'. This can be done in the following way. Let A_0 be the symbol in the first cell of M's tape, A_{-1} be the symbol in the cell to the left of it, A_{-2} the symbol to the left of that one and so on (Figure 11.5a). Similarly, let A_1 be the symbol in the cell to the right of the first cell, A_2 the symbol in the cell to the right of that one and so on. We can represent such a tape with a one-way infinite tape of M', where the cell i will keep *a pair* of symbols from the symmetrical cells of the M's tape. Thus, the first cell of M''s tape will contain the pair of symbols $(A_0, \oslash)$, where $\oslash$ is a special symbol which is not in the tape alphabet of the two-way infinite tape TM, and that is used to denote the border between the portion of the tape to the left of the first cell in M's tape and the portion of the tape to the right of the first cell in the same tape. The second cell is then going to contain the pair of symbols (A_1, A_{-1}), the third cell the pair of symbols (A_2, A_{-2}) and so on (Figure 11.5b).



(a) A Two-Way Infinite Tape Turing Machine M With Symbols $\ldots, A_{-3}, A_{-2}, A_{-1}, A_0, A_1, A_2, A_3, \ldots$ on the Tape.



(b) The Basic Turing Machine M' Simulating the Above Two-Way Infinite Tape TM.

We can also interpret the tape of M' as divided into upper and lower portion. Upper portion contains the symbols in the cells of M to the right of the first cell, while the lower portion contains the symbols in the cells of M to the left of the first cell. Each cell contains the upper part (the first component of the pair of symbols in it), and the lower part (the second component of the pair of symbols in it). M' can then simulate M by operating only on the upper portion of its tape and mimicking exactly the transitions of M when M is reading the cells to the right of the first cell. When M is reading the cells to the left of the first cell, M' operates only on the lower portion of its tape and moves the tape head in the *opposite* direction of M. To distinguish between these two modes of operation, M' can have two states corresponding to each of the states of M. For a state q of M, M' can have state q_U where it is operating on the upper portion of its tape, and state q_L where it is operating on the lower portion of its tape. The switch between the two modes is done when M makes move from the first cell to the left (this makes M' change its mode from operating on the upper portion of the tape to operating on the lower portion of the tape), or when M makes the move from the cell on the left of the first one to the first cell (this makes M' change its mode from operating on the lower portion of the tape to operating on the upper portion on the tape). For example, if there is transition $\delta(q, X) = (p, Y, L)$ for M, this will correspond to two transitions of M': $\delta'(q_U, (X, E)) = (p_U, (Y, E), L)$ for any tape symbol E (this is the move where M' operates on the upper portion of the tape, replaces the symbol X in the upper part of the scanned cell with Y, leaves the symbol in the lower part of the cell unchanged, and moves its tape head in the same direction as M) and $\delta'(q_L, (E, X)) = (p_L, (E, Y), R)$ (this is the move where M' operates on the lower portion of the tape, replaces the symbol X in the lower part of the scanned cell with Y, leaves the symbol in the upper part of the cell unchanged, and moves its tape head in the opposite direction to M). The exception will be the moves on the symbol of type $(E, \oslash)$, as they may change the mode of operation of M' (i.e. switch state from q_U to p_L or from q_L to p_U). We will omit the details of constructing precisely the transition function δ', the definition of M' with this transition function and the proof that M and M' accept the same language. □

11.2.2 Multitape Turing Machine

Another important modification of the basic Turing Machine model is a *multitape Turing Machine*. A Multitape Turing Machine has multiple tapes, each of which has its own tape head (See Figure 11.6).

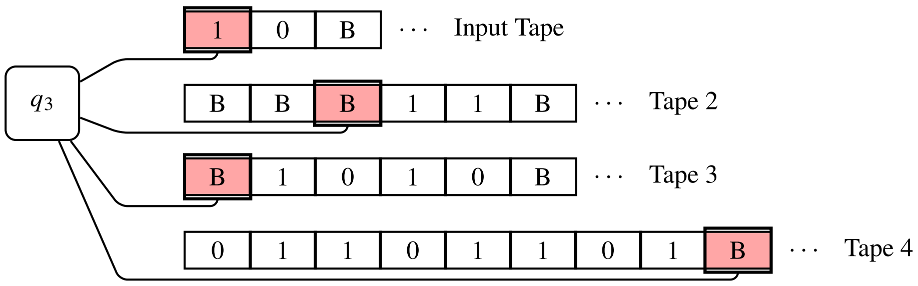


Figure 11.6: A 4-Tape Turing Machine

The machine is in one state at a time. In each move, the machine reads all the symbols in the cells scanned by type heads, overwrites each of these symbols with new symbols, moves each of the tape heads and goes to the new state. The transition function for the multitape Turing Machine with n tapes is of the form

$$\delta(q_0, X_1, X_2 \ldots, X_n) = (p, [Y_1, D_1], [Y_2, D_2], [Y_3, D_3], \ldots, [Y_n, D_n]).$$

This means that the machine in the state q and scanning the symbol X_1 on the first tape, X_2 on the second tape and so on, moves to the state p, replaces the symbol X_1 with Y_1 on the first tape and moves the tape head of that tape in the direction D_1, replaces the symbol X_2 with Y_2 on the second tape and moves the tape head of that tape in direction D_2 and so on.

One tape of the machine is the input tape. The multitape Turing Machine starts the computation on string w with string w on its input tape, and with all the other tape containing just blank symbols. The string is accepted if the machine at some point during its processing enters one of the accepting states.

Theorem 11.2: Equivalence Between Multitape Turing Machines and Basic Turing Machines

The class of languages accepted by the Multitape Turing Machines is the same as the class of languages accepted by basic Turing Machines.

The process of simulating a multitape Turing Machine by a basic, single-type Turing Machine is conceptually simple, but quite tedious to describe precisely. It is described in some detail in the "Multitape Turing Machines" video. We are going to omit the details here.

11.3 Turing Machines as Computers of Integer Functions

So far, we have seen Turing Machines as language acceptors, in that they would receive a string as an input, and would produce "yes" or "no" as an answer (depending on whether the input string is accepted or not), or would alternatively loop forever. But, thanks to the fact that the Turing Machine can write on its tape, Turing Machines can also be seen as computers of functions with integer arguments. Here we are going to focus only on functions that accepts non-negative integers and return a non-negative integer, but this is easily extended to negative integers too. The traditional (and very inefficient) approach is to represent integers in unary representation - the integer number $i \geq 0$ is encoded by the string 0^i. Thus, 0 is encoded by ϵ, 1 is encoded as 0, 2 is encoded as 00 and so on. In this way, we can encode every integer number as a string of 0's. For functions that accept tuples of integer numbers, e.g. $(i_1, i_2, \ldots, i_k)$, we can encode each element of the tuple and then concatenate all these encodings together, separating them by a symbol 1. Thus, the tuple $(i_1, i_2, \ldots, i_k)$ can be encoded as $0^{i_1}10^{i_2}1 \ldots 10^{i_k}$. For example, the pair $(3, 2)$ can be encoded as 000100. In this way, we encode tuples of integer numbers as strings over the alphabet $\{0, 1\}$. Encoding of an input to