Chapter 12

Undecidability

In the Chapter 11, we have seen the definition of Turing Machines. We have also seen that the Turing Machines as a computing model are powerful enough to be used both for language recognition, in the same way as we used finite automata, but also as computers of integer functions. We have also seen some reasonably simple Turing Machines to recognise languages that were beyond capabilities of finite automata. Finally, we have also seen some modifications of Turing Machine that do not increase the power of the model. All this gives indication that the Turing Machine is a much more powerful model of a computer than any form of finite automata. In fact, in Section 12.1, we will argue that Turing Machines are as powerful as computers themselves. This is known as Church's Thesis.

One feature that makes Turing Machines especially powerful is that they can be encoded as binary strings. This allows us to have a Turing Machine that will accept another Turing Machine and an arbitrary string as an input, and which will simulate the processing of that string by that Turing Machine. This is equivalent to the computer that receives a program and some input to the program, and which runs that program on that input. Therefore, Turing Machines are not limited to performing one single operation (such as, for example, adding two integer numbers), but can be used to receive a description of a computation as an input. We show one way of encoding a Turing Machine in Section 12.2.

Being able to encode Turing Machines as binary strings has one more very important application. Using these encodings, we can show that there is only countably many Turing Machines. But, crucially, we can use encodings of Turing Machines to construct a language that is not Turing recognisable (or partially decidable). Taking into account the equivalence between Turing Machines and computers, this shows that there exist problems that are not solvable by a computer, no matter how much time or space we have. This is the first grand result of the Theory of Computation that we are going to prove, and we do this in Section 12.3.

Finally, the result that there exists a language that is not recognisable can be used to prove the second grand result of the Theory of Computation - that there exist languages that are recognisable, but not totally decidable. This means that there are languages for which there exist Turing Machines that recognises them, but no such Turing Machine halts on all inputs. This is shown in Section 12.4.

12.1 Church's Thesis

Turing Machines have shown to be so powerful that it is reasonable to believe they can be used to compute every function that conforms to the intuitive notion of a "computable function". That is, that every function that can be computed using, for example, a computer can also be computed using a Turing Machine. This is known as *Church's Thesis* (or sometimes as *Church-Turing Thesis*). Since this thesis relies on an intuitive notion of a computable function, it is impossible to prove or disprove it. But what gives us confidence in the truth of the thesis is that the the class of partially recursive functions is also the class defined by many other formalisms such as λ-calculus, general recursive functions and random access machines (RAMs). The fact that so many different models define exactly the same class of functions makes us suspect that this class of functions is very important. Furthermore, the fact that modern computers in their essence cannot really do anything significantly more complex than can Turing Machines strengthens the belief in the correctnes of the Church's Thesis.

One can ask how is it possible that seemingly simple model as Turing Machine can calculate every function that a computer can. In essence, every computer consists of a processor with internal memory (RAM) and external storage. Processor roughly corresponds to the finite state control, and internal memory

and external storage can be simulated using the tape. Therefore, every key component of a computer can be simulated by a Turing Machine. The simulation will, naturally, be very inefficient, in that even the simplest operations, that a computer can do very quickly, take many steps with a Turing Machine, but currently we are only interested in whether a Turing Machine can simulate a computer, and not how efficiently this is done. A consequence of this capability of Turing Machines is that a problem solvable by a computer is also solvable by a Turing Machine and vice versa. In this way, to find a problem that is unsolvable by a computer, we need to find the problem that is not solvable by a Turing Machine. This is what we will do in the following sections.

12.2 Encoding Turing Machines as Binary Strings

In this section, we will show how a Turing Machine can be encoded as a binary string. This will allow us to order Turing Machines into a sequence, which is a key step to show both that there exists a language that is not recognisable, and to show that there exist a language that is recognisable, but not totally decidable. These are two key results in the Computability theory.

We are going to focus on Turing Machines that operate over the input alphabet $\{0, 1\}$, as this makes it easier to encode Turing Machines as binary strings. One result result that helps us with devising a method to encode Turing Machines as strings is that if a language L over the alphabet $\{0, 1\}$ is accepted by some Turing Machine, then it is also accepted by a Turing Machine that has tape alphabet $\Gamma = \{0, 1, B\}$. Therefore, we only need to consider Turing Machines with the tape alphabet $\{0, 1, B\}$. Furthermore, we only need to consider Turing Machines that have a single final state, because every Turing Machine can easily be transformed into a Turing Machine with a single final state without changing the language accepted.

Therefore, let us assume we are given a Turing Machine $M = (Q, \{0, 1\}, \{0, 1, B\}, \delta, q_1, B, \{q_2\})$, where the set of states is $Q = \{q_1, q_2, \ldots, q_n\}$. Since for each such machine, the input alphabet, tape alphabet, start state, blank symbol and final state are the same, the only difference between different machines is the set of states they have and the transition function. Furthermore, of these two, the set of states is implicitly given in the transition function. Therefore, all that we need to encode for M is its transition function δ. We do this by encoding each transition as a binary string. Let us denote by X_i the tape symbols of M. That is, $X_1 = 0, X_2 = 1, X_2 = B$. We can encode the tape symbol X_i by i 0's (similarly as we encoded nonegative integer numbers when we considered Turing Machines as computers of integer functions). In the same way, we can encode any state q_i by i 0's. Finally, let us denote directions in which the tape head can move by $D_1 = L$ and $D_2 = R$ and let us encode D_i by i 0'S. Let us then assume that there is a transition

$$\delta(q_i, X_j) = (q_k, X_l, D_m).$$

This transition can be encoded by encoding, in turn, q_i, X_j, q_k, X_l and D_m, and separating encodings by a single 1. Thus, this transition can be encoded by

$$0^i 10^j 10^k 10^l 10^m.$$

To encode the whole transition function, we simply encode every transition in it and we separate the encodings of transitions by 11. Therefore, if e_1 is the encoding of first transition, e_2 encoding of the second transition and so on, with e_p the encoding of the last transition, then the encoding of the whole transition function is

$$e_1 11 e_2 11 \ldots 11 e_p.$$

And with this, we are done. To mark the start and the end of the encoding of the complete transition function, we use the strings 111. Therefore, if e is the encoding of the transition function δ, the encoding of M will be

$$111e111.$$

We have found a binary string that encodes an arbitrary Turing Machine M with the properties that we require (tape alphabet $\{0, 1, B\}$, start state q_1, final (accepting) state q_2).

Note that the order in which we have decided to encode transitions is arbitrary. Therefore, the same Turing Machine M can have many different encodings, depending on the order in which we encode transitions. However, given a valid encoding of a Turing Machine, we can be sure that such encoding describes exactly one Turing Machine.

12.2.1 Standard Enumeration of Binary Strings

For the purposes of constructing a language that is not partially decidable, we will need to somehow order Turing Machines, based on their encoding. To do that, we first need to establish ordering of binary strings. This is fairly easy to do, as we can introduce ordering of strings based on their length, with the strings of the same length being put in lexicographical order (with symbol 0 coming before symbol 0). Thus, we can put all the binary strings in an ordered sequence

$$(\epsilon, 0, 1, 00, 01, 10, 11, 000, 001, 010, 011, 100, 101, 110, 111, 0000, \dots).$$

Can we determine the place in this sequence in which some string w appears? Of course we can. Each binary string can be seen as a positive integer number in the binary system. The place of string w in the above sequence is exactly the number which is represented by w in the binary system, with the caveat that we need to add 1 at the start of w (otherwise, for example, 0, 00, 000 and so on all represent the number 0, but they are obviously at different places in the sequence). Thus, for example, the string ϵ is at the place which is the number $1\epsilon = 1$ in the binary system, which is 1. The place of the string 000 is the number 1000 in the binary system, which is 8. We can, indeed, verify that 000 appears in the eight place in the above sequence. In the further discussion, unless specified otherwise, we will denote by w_i the i-th string in the above sequence. Therefore, $w_1 = \epsilon$, $w_2 = 0$, $w_3 = 1$ and so on.

With this, we have introduced ordering of binary strings, and therefore we can compare the two binary strings and say that one of them comes before the other in this ordering. Since encodings of Turing Machines are binary strings, we can also introduce ordering of Turing Machines over the input alphabet $\{0, 1\}$. It is useful to consider any binary string as an encoding of a Turing Machine. If the string is not a valid encoding, i.e. if it does not start and end with 111 or if it does not contain encodings of valid transitions separated by 11's, then we imagine this as the encoding of a Turing Machine that makes no moves, and as such accepts the $\emptyset$ language. Thus, we can order all Turing Machines over the input alphabet $\{0, 1\}$ as $M_1, M_2, M_3, \dots$, where M_i is the Turing Machine encoded by the binary string w_i that is i-th in the ordered sequence. For example, M_4 is the Turing Machine encoded with the string 00. This is obviously a Turing Machine that has no moves.

In this ordering of Turing Machines, there will be many Turing Machines that accept the same language, and many Turing Machines will be exactly the same. For example, whenever we encode the transitions of some Turing Machine in different order, we get the different encoding and different M_i, even though the actual Turing Machine is the same. This does not present any problem for us.

Example of Turing Machine Encoding

Consider a Turing Machine $M = (Q, \Sigma, \Gamma, \delta, q_1, B, \{q_2\})$ such that $Q = \{q_1, q_2, q_3\}$, and Γ is, as usual, $\{0, 1, B\}$. The transition function δ is given by the table

δ	0	1	B
q_1	$(q_2, 0, R)$	-	(q_2, B, R)
q_2	-	-	-
q_3	-	-	(q_2, B, L)

This Turing Machine has three transitions. Let us encode each of them:

- The transition $\delta(q_1, 0) = (q_2, 0, R)$ has encoding 01010010100;
- The transition $\delta(q_1, B) = (q_2, B, R)$ has encoding 010001001000100;
- The transition $\delta(q_3, B) = (q_2, B, L)$ has encoding 0001000100100010;

Therefore, the one encoding of the transition function δ is

$$01010010100110100010010001001100010001001000010.$$

The encoding of the whole M is

$$11101010010100110100010010001001100010001001000010111.$$

This is the machine $M_{17246064437922071}$. Note that if we encoded the transitions in different order, we would get a different encoding. For example, if we first encode the second transition, then the first, we would get the code

$$1110100010010001001101010010100110001000100100010111.$$

This is the machine $M_{4091365757061399}$ - the different encoding that describes exactly the same Turing Machine.

12.3 Diagonalisation and A Language That is Not Partially Decidable

Now we are ready to give an example of a language that is not partially decidable. Finding such a language is, as we will see, far from trivial, because Turing Machines are far more powerful than finite automata and can recognise a much wider range of languages. Therefore, our first language that is not partially decidable will be quite artificial in a way.

In the previous section, we have seen how we can order, or enumerate, all Turing Machines over the input alphabet $\{0, 1\}$ into the sequence $(M_1, M_2, M_3, \dots)$. The languages accepted by these Turing Machines are $(L(M_1), L(M_2), L(M_3), \dots)$. This sequence will have many elements that are repeated, but we know for sure that it definitely contains all partially decidable languages over the alphabet $\{0, 1\}$. Because if the language L over the alphabet $\{0, 1\}$ is partially decidable, then there exists a Turing Machine with the input alphabet $\{0, 1\}$ that recognises/accepts that language. Then there also exists the encoding of that Turing Machine, therefore that Turing Machine is some M_i from our sequence of Turing Machines. Hence, L is then exactly the same as $L(M_i)$. Therefore, our goal is to find the language L that is not in this sequence of languages.

To find L, we are going to use the process called *diagonalisation*. The similar process is used, for example, in the proof that there are more real numbers than integer numbers. We are going to create an infinite "table", where the i-th row corresponds to the Turing Machine M_i, the j-th column corresponds to the string w_j, and entry at the position (i, j) is either "yes", in the case that M_i accepts w_j, and "no" if M_i does not accept w_j (regardless of whether it rejects it or it never halts).

	w_1	w_2,	w_3	w_4	w_5	...
M_1	**no**	yes	no	yes	yes	...
M_2	yes	**yes**	yes	no	no	...
M_3	yes	no	**yes**	no	yes	...
M_4	no	yes	no	**no**	yes	...
M_5	yes	no	yes	no	**yes**	...

Let us now construct our language, and let us call it L_d. For each string w_i, we decide whether to include it in L_d or not. The string w_i will be in L_d if M_i does not accept w_i. So, w_1 will be in L_d if M_1 does not accept w_1, w_2 will be in L_d if M_2 does not accept w_2 and so on. This corresponds to looking just at the entries in the above table along the diagonal (hence the name diagonalisation), and adding the string to L_d only if its entry in the diagonal is "no". Therefore, our language L_d can be described as

$$L_d = \{w_i | M_i \text{ does not accept } w_i\}.$$

Let us now assume that L_d is partially decidable. Then there exists some Turing Machine M that recognises it and, recalling our previous discussion, this Turing Machine must be some Turing Machine M_j. Therefore $L(M_j) = L_d$. Let us now ask whether the string w_j belongs to L_d. Clearly, there are only two options - it either belongs to L_d or it does not. Let us consider these two options now:

- Let $w_j \in L_d$. Then, by the definition of L_d, this means that M_j does not accept w_j, because the string is in L_d only if it is not accepted by the Turing Machine that it encodes. On the other hand, since $L(M_j) = L_d$, w_j belongs to the language recognised by M_j, therefore M_j has to accept w_j. So, M_j both has to accept and to not accept w_j, which is a contradiction.
- On the other hand, let $w_j \notin L_d$. Then, since $L(M_j) = L_d$, this means that M_j does not accept w_j. But since the definition of language L_d is the set of all strings such that the Turing Machine that they encode does not accept them, then w_j has to belong to L_d. Which again means that M_j has to accept w_j, because $L(M_j) = L_d$. We again reach a contradiction.

So, w_j can not belong to L_d, but it also cannot not belong to L_d. This is obviously impossible. Therefore, the assumption that there exists a Turing Machine M_j that recognises L_d leads us to contradiction. Which means that this assumption must be false and no Turing Machine from our sequence of Turing Machines can accept L_d. Since this sequence contains all the Turing Machines over the input alphabet $\{0, 1\}$, this means that no Turing Machine can recognise L_d. This means that L_d is not recognisable (or partially decidable). With this, we have proven our first crucial result about computability, which is summarised in the following theorem.

Theorem 12.1: Language that is not Partially Decidable/Recognisable

Let

$$L_d = \{w_i | M_i \text{ does not accept } w_i\},$$

where w_i is the i-th string in the ordering of the strings over $\{0, 1\}$ and M_i is the Turing Machine with the encoding w_i. Then L_d is not partially decidable.

12.4 Universal Turing Machine and A Language That is Not Totally Decidable

In the previous section, we have found a language that is not partially decidable or recognisable. This means that the class of languages that are not recognisable is not empty. We also know that the class of languages that are totally decidable is not empty - there are many many languages for which Turing Machines exist that halt on every input and that accept exactly that language. The only class of the languages that we have considered, and for which we still haven't concluded whether it is empty or not is the class of paritally decidable but not totally decidable languages. As a reminder, a language is in this class if there exists a Turing Machine that recognises it, but no Turing Machine that recognises it halts on all inputs. We know certainly that every totally decidable language is also partially decidable, but we don't know whether there are any partially decidable languages that are not totally decidable. Therefore, we don't know whether the class of partially decidable and the class of totally decidable languages are really different. In this section, we are going to prove that they, indeed, are different.

Our strategy for achieving that goal will be to first construct a Universal Turing Machine, which will be a Turing Machine that accepts arbitrary other Turing Machine and a string, and simulates the operation of that Turing Machine on that string. We are then going to consider the Universal Language, the language of all encodings $\langle M, w\rangle$ of a Turing Machine M and a string w, such that M accepts w. We are going to show that the Universal Turing Machine accepts the Universal Language, therefore that language is partially decidable. Then we are going to take a slight detour and use the concept of the Universal Turing Machine to provide simple proofs of some important properties of partially decidable and totally decidable languages. Finally, we are going to use some of these properties to show that no Turing Machine can decide the Universal Language, which is to say that no Turing Machine can halt on all inputs and accept exactly the strings from the Universal Language. Thus, the Universal Language is partially decidable but not totally decidable.

In most of the literature, the languages that are not totally decidable are called *undecidable*. We will also sometimes use this terminology. Note that we do not make an assumption that the undecidable language is even partially decidable. All we know is that it is not totally decidable. Also, if language L is accepted by Turing Machine M that halts on all inputs, we will say that M *decides* L.

12.4.1 Universal Turing Machine and Universal Language

As we said above, the Universal Turing Machine is a regular Turing Machine that is able to simulate an arbitrary Turing Machine M over the language $\{0, 1\}$ on an arbitrary string over $\{0, 1\}$. We can assume that M has only three tape symbols, 0, 1 and B, that the starting state is q_1 and the only accepting state is q_2. We have seen in Section 12.2 how such a Turing Machine can be encoded as a binary string. If we concatenate an arbitrary string w to the encoding of M, we get a binary encoding of a Turing Machine M and a string w. We will denote this encoding by $\langle M, w\rangle$. We can then construct the Universal Turing Machine U that will run the Turing Machine M on input string w.

For simplicity, we will assume that U has four tapes (Figure 12.1). We have seen that adding additional tapes does not increase the power of the model - exactly the same class of languages is accepted by single-tape Turing Machines as with multi-tape ones. The first tape holds the input $\langle M, w\rangle$. The second tape simulates the tape of M. The third tape holds the state of M. The fourth tape is an auxiliary scratch tape, used to help with simulation. The fourth tape is not strictly necessary, but makes our life a bit easier. At start, the second