4 Computability

In this section we leave the theme of a hierarchy of languages and their corresponding automata behind. (That's why there was a summary for Sections 1 to 3.) However, we will still meet some of the concepts, and a few languages, as examples in Section 4. The aim in what follows is to examine Turing machines as abstract computation devices. It should be quite clear from what we have seen so far that Turing machines can be applied to many more tasks than deciding, recognizing and enumerating languages.

We can think of a Turing machine as a *program* which takes one or more inputs which are written on the tape before the execution of the program starts.[23] Now the execution of the program may either finish, in which case the content of the tape can be considered the output of the program on the given input, or it may continue forever, in which case there is no output.

The topics we consider in this section are the following:

- What kind of things can a Turing machine compute?
- What happens if we vary the concept of a Turing machine?
- How powerful are Turing machines compared with other devices, in particular 'real' computers?
- Are there problems that cannot be solved by any computing device (including Turing machines?)
- What kinds of problems are solvable, and what kinds aren't?

4.1 Turing machine and computations

Turing machines were created originally to define formally the idea of computing something using a set of instructions, or an *algorithm*. This was in the thirties, before sophisticated computation devices had been developed[24], and Turing envisioned these instructions to be carried out by a person. The question of recognizing various languages generated by grammars did not arise until quite some time later. Originally Turing was thinking of computations on natural numbers for the most part.

So what kind of 'computations' does a Turing machine carry out? There are a number of different ways of formalizing that. What they all have in common is that since the Turing machine has a memory device it can write on (the tape), (some of) what is on the tape at the end of a computation can be considered the output of the machine for the given input.

Here is an example. We want to construct a machine which goes to the right end of its input string, prints an a in the following cell and then stops with the head pointing immediately to the right of that cell. We need only two states for that. Assume that the tape alphabet is $T = \Sigma \cup \{\sqcup\}$. Here is the transition function δ for the machine.

[23] Clearly we have to ensure that inputs are separated appropriately, either by blanks or by other suitable symbols such as #.

[24] The most elaborate machines at the time were With the exception of Babbage's 'engines', purely mechanical devices.

δ	$x \in \Sigma$	$\sqcup$
0	$(0, x, R)$	$(1, a, N)$
1	stop	stop

When run on the input string *aabba* the machine will run through the following configurations.

$$0aabba \to a0abba \to aa0bba \to aab0ba \to aabb0a \to aabba0 \to aabbaa1.$$

We could interpret the result of this computation to be the input string with an a appended. Note that when we consider Turing machines as computation devices which take an input and produce an output on the tape we are often not really interested in acceptance of input strings, and in that case we do not specify any accepting states.

As another example, a TM can copy its input string. Here is an example where $\Sigma = \{a, b\}$ and $T = \Sigma \cup \{\sqcup\}$. The machine leaves one blank before starting the copy. If the input string is a blank it does nothing since in that case the tape can be assumed to be entirely blank.

δ	a	b	$\sqcup$
0	$(1, \sqcup, R)$	$(2, \sqcup, R)$	stop
1	$(1, a, R)$	$(1, b, R)$	$(3, \sqcup, R)$
2	$(2, a, R)$	$(2, b, R)$	$(4, \sqcup, R)$
3	$(3, a, R)$	$(3, b, R)$	$(5, a, L)$
4	$(4, a, R)$	$(4, b, R)$	$(6, b, L)$
5	$(5, a, L)$	$(5, b, L)$	$(7, \sqcup, L)$
6	$(6, a, L)$	$(6, b, L)$	$(8, \sqcup, L)$
7	$(7, a, L)$	$(7, b, L)$	$(0, a, R)$
8	$(8, a, L)$	$(8, b, L)$	$(0, b, R)$

This machine might be more complicated than you thought it would be, because the only actions a Turing machine can carry out are fairly simple. Here is a description for how the machine works.

1. Look at the current symbol, remember whether it's an a or a b (by going into state 1 or 2), overwrite it with a blank, move to the right. If it's a blank, stop.
2. Move to the right until a blank is found. This is the end of the input string.
3. Move to the right until a blank is found. This is the end of the copy so far. Write the remembered letter, move one to the left. (Keep remembering which letter it was that is being copied, again by using different states for different letters, so that it can be rewritten in its old place.)
4. Move to the left until a blank is found. This is the one between the two strings.
5. Move to the left until a blank is found. That's where the last letter copied originally was. Write the remembered letter, move one to the right. Start over at 1.

The one clever idea for this machine is to 'remember' where it had got to in copying the input string by replacing the 'current character' by a blank, which gets overwritten by the

original character before moving to the next letter. Clearly we can build this machine from simpler parts which perform tasks like 'go to the right until you find a blank' and 'write some character, then move to the left/right'. It is not difficult to put Turing machines together in that fashion.

Consider the two machines given by the following transition tables. Call the left one M_1, the right one M_2. They are both very simple.

δ_1	a	b	$\sqcup$
0	$(0, a, R)$	$(0, b, R)$	stop

δ_1	a	b	$\sqcup$
0	$(1, a, R)$	$(1, a, R)$	$(1, a, R)$
1	stop	stop	stop

Machine M_1 moves to the right until it finds a blank, while M_2 writes an a and moves one step to the right. We can now create a machine which first behaves like M_1 and then (when it has reached the position of the first blank) like M_2, that is, it prints an a there. All in all we get a machine that moves to the right until it finds a blank and prints an a into that cell, and then moves one step to the right of that cell. All we have to do is

- rename the states of machine M_2 (so that they are different from the names of the states of machine M_1);
- replace the stop command(s) in machine M_1 by transitions to the start state of machine M_2 (without change of head position).

In other words, we change the machines as follows.

δ_1	a	b	$\sqcup$
0	$(0, a, R)$	$(0, b, R)$	$(1, \sqcup, N)$

δ_1	a	b	$\sqcup$
1	$(2, a, R)$	$(2, a, R)$	$(2, a, R)$
2	stop	stop	stop

This allows us to bring it all together in one machine which we name M_1M_2 whose transition function is

δ	a	b	$\sqcup$
0	$(0, a, R)$	$(0, b, R)$	$(1, \sqcup, N)$
1	$(2, a, R)$	$(2, a, R)$	$(2, a, R)$
2	stop	stop	stop

This machine will now go to the right until it finds a blank and print an a in the cell to the right of that blank. Exercises 27 and 28 invite you to design a few Turing machines.

There is a lot more information of how to build Turing machines, as well as a graphical description, in [HMU01], Chapter 8.2.

4.2 Variants of Turing machines

We have already seen that there are variants of Turing machines. One can define Turing machines without accepting states, as we have argued above. When we moved from PDAs to Turing machines we gave the machine a tape, which means random-access memory. Does anything change if we change the nature of this memory? How about non-deterministic Turing machines?

One-sided tapes. Nothing is lost or gained if we restrict our infinite tape to one that is infinite on one side only. In that case it is usually assumed that the machine starts with the head pointing at the first cell, and that the input string starts 'at the beginning' of the tape. Let us assume that the tape stretches to infinity on the right. A move to the left from that first cell results in no action at all. These Turing machines have precisely the same power as those we defined. Given a Turing machine we can simulate its action via a Turing machine with a one-sided tape as follows. Add to the tape alphabet T a special symbol, say # (which is different from $\sqcup$ and different from all elements of Σ).[25] We then split up the one-sided tape into blocks as we go along, some blocks simulating 'the right side of the tape' and the others simulating 'the left side of the tape'. We use # to separate these blocks. Alternatively, we can do the following. Assume all the cells of the one-sided tape are numbered. Then we can use the even numbered cells to contain the content of the cells to the right of the head when the machine starts, and the odd numbered ones for the cells to the left. In either case, changing the transition table for the original Turing machine to one that works with a one-sided tape can be very fiddly, and usually requires quite a few more states.

Multiple tapes. Alternatively there can be several tapes, each with its own read/write head. Then at each stage of a computation, the next action is determined by the content of the cells at which the heads point (one for each tape) as well as the current state. That next action consists of one write and move action for each head. Such machines certainly are more complicated to describe, but then, they make some things a lot easier. The copy machine which we describe above, for example, becomes very straightforward if two tapes are available. Just let the first tape contain the string to be copied, and assume that the copy is to appear on the second tape (which we assume to be empty at the start).

In order to give a transition table for this machine, we have to allow it to depend on the content of *both* tapes. As with PDAs, we use a blank entry in case when a particular argument doesn't matter. There is a pair of instructions, one for the first tape and one for the second tape. A table defining such a copy machine might look like this:

δ	$(a, \ )$	$(b, \ )$	$(\sqcup, \ )$
0	$(0, a, R, a, R)$	$(0, b, R, b, R)$	stop

Therefore multi-tape machines can be more convenient. But can they actually do more than single tape ones? The answer is no, and the argument is very similar to the one we used before to go from one-sided to ordinary Turing machines.

We will give it here for simulating a two tape machine with a one tape machine. Again we use blocks on the one tape to simulate the two tapes, separating them by #. However, we also need to keep track of the position where the two heads are from one step to the next. We do this by using only every other cell to store actual data, thus leaving space to mark a cell as having the head there. We can use any symbol different from $\sqcup$ and # which does not occur in Σ for that, say ! (exclamation mark). A snapshot of such an encoding is drawn in Figure 32.

The machine then runs at follows. It has to look at *both* current head positions (necessitating it to move the head just to look up the current symbols!), and then mimics the action of each head one after the other. At the same time it has to move the ! symbols as appropriate (that is, erasing the existing one and writing another to mark the new head position). Hence

[25] This is why we allowed the tape alphabet to contain symbols not in the original alphabet Σ.

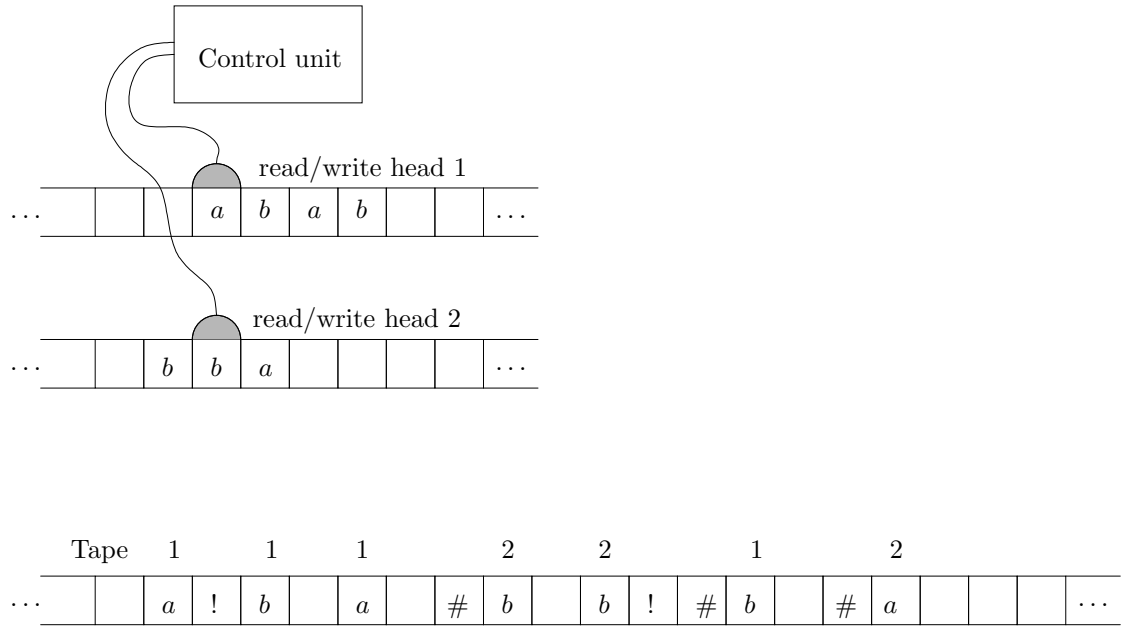

Figure 32: Simulating a two-tape Turing machine.

for each one step the multi-tape machine makes, the single tape machine must make several. As the tape fills up, the machine has to move further and further to find empty space to work on. Hence the simulation is very slow, but we are not concerned with speed, only with the power to do the same computations. An alternative way of doing this is to split the tape into blocks of two from the start (then no symbol is required to distinguish between the two), the first of which holds any data, and the second is again used to mark the position of a head. Then every other block corresponds to the same tape. (If we numbered the blocks, we would split them into those with even and those with odd numbers.)

Note that the equivalence between single-tape and multi-tape machines means that we may use the latter for any proof where it is more convenient to do so. In particular if we would like to build a new Turing machine which runs two previous machines 'in parallel' (on the same input string), we can easily do so by using a machine with several tapes. (We need two copies of the input string in case one of the machines interferes with it, and two tapes for the machines to work on.) This is very handy for building new machines from existing ones.

Multi-dimensional tapes. We can make our machine even more flexible by giving it a multi-dimensional tape and allowing it to move its head in each direction. For example if the tape is two-dimensional then the head can move left, right, up and down. This offers even more space than a multi-tape machine: infinitely many cells in each direction. Again this does not increase the power of the machine, but simulating it with a one-tape machine becomes somewhat more complicated. For this to work it is crucial that at any given point in time, the machine can only have made use of finitely many cells, that is finitely many rows and columns. Exercise 29 asks you to think about representing a two-dimensional tape, making use of only one dimension.

Non-deterministic Turing machines. We have already seen non-deterministic finite au-

tomata and non-deterministic PDAs. We can similarly define non-deterministic Turing machines where again we replace the transition function by a transition relation. That means that given an input string, there are possibly several different computations that may be carried out when the machine starts. Again we define acceptance via the existence of at least one computation where the machine stops in an accepting state.

Unlike the case of PDAs, this does not increase the power of the device. Every non-deterministic Turing machine can be simulated by a deterministic one. We can view the different computations a machine might carry out as a tree, with a branch whenever there are several possibilities for the machine to behave. Note that the resulting tree is finitely branching, that is, at every point where it branches there are only finitely many new branches. A deterministic machine can check these computations by traversing this tree breadth-first. The easiest way of describing such a simulation uses three tapes (which we may do since we can simulate the result on a one-tape machine), one to store the input, one to mirror the computations, and one to keep track of the current location in the tree of non-deterministic computations. We will not go into more detail here, see for example [Sip97] pp. 138f., [HMU01] pp. 341f.

In what follows, we are mostly interested in what a Turing machine can do *in principle*. Hence we feel free to have our machines to be any of the variants listed above, knowing that the result can be simulated with a one-tape Turing machine. It turns out that the most useful of these in practice is the ability to have more than one tape.

4.3 The power of Turing machines—the Church-Turing Thesis

So what can Turing machines actually do? You may think that Turing machines cannot be very powerful, given their limited set of actions. But just how limiting is that? Leaving aside the fact that computers can only have limited memory, it is clear that we can simulate the action of a Turing machine on a computer. If you search the web you will find a number of on-line simulators to play with. So it seems clear that (apart from the memory issue, of course) Turing machines are less powerful than computers.

But is that really true? Certainly high level languages such as Java or even C seem to be a far way removed from having to worry about all the small steps that a Turing machine carries out. But if we think about it more carefully we remember that Java code is actually compiled into machine code before it is executed. Machine code certainly is much less sophisticated and has a much reduced set of instructions.

Such computations are often modelled using the idea of a *von Neumann machine* with a very limited set of instructions, and where memory cells are of a fixed size. All the instructions used in that machine take one or two arguments (the contents of certain registers, or memory cells), and return a value to be written into such a cell. You have seen examples of such languages in CS101/CS100. We can take these instructions and turn them into a transition table for a Turing machine (one instructions may require several lines in the transition table) and thus come up with a (fairly large) transition function. Clearly, converting registers in the machine's memory into cells on a Turing machine's tape is straight forward. The von Neumann machine can access its memory cells at random by assigning a name to each cell. We can model this by using a second tape to keep track of which named registers are stored in which cells. We can then manoeuvre among those simulated registers by reading off from the second tape where we need to go. This gives us a Turing machine (although admittedly we have only described it in a fairly hand-wavy manner) which simulates a von Neumann

machine, and thus a real computer. In other words, a Turing machine can compute precisely the same things as a computer with arbitrarily large memory![26]

There are other models of computation than von Neumann machines and Turing machines, many more, in fact. As long as they are machines of some kind, we can argue that they are equivalent provided that they can simulate each other. An entirely different approach to computation can be found in mathematics. It does not mention any computing devices at all. It is restricted to functions which take natural numbers as inputs and give natural numbers as output. Such a function is defined to be *computable* if it can be built from primitive functions such as the successor function, the test for 0 (and projections from a tuple of natural numbers to one of those numbers) using the composition of functions and *recursion*.[27]

It turns out that all these models of computation result in *precisely* the same notion of what is computable! That lead the American logician Alonzo Church and his English colleague Alan Turing to postulate that there is only one sensible notion of computability, and that no matter how many different models are developed, they all lead to the same thing. Clearly this is not a mathematical conjecture that can be proven, it is much too vague for that. We cannot hope to show that in the future, somebody will not come up with a notion that does differ from all those that went before. Given the time and effort that has gone into the investigation of such models however, the Church-Turing Thesis seems to be at least plausible. What we do know, on the other hand, is that something that is not computable using, say, Java programs will not be computable using any other existing computing device. In that sense Turing machines, the earliest such model, capture the very essence of computation (at least to the best of our knowledge today). Unfortunately there is no time in this course to go into this in any detail. Exercises 30 and 31 introduce other (theoretical) models of computation which have the same power as Turing machines.

In what follows below we will therefore sometimes switch the notion of computability under consideration: We start by having a look at programs in C and Java, and then demonstrate how an analogous result can be obtained for Turing machines. The Church-Turing Thesis tells us that it does not really matter which notion we study—they all lead to the same results. Theoreticians typically prefer to deal with Turing machines because arguments involving these can easily be made (mathematically) rigorous. There is no worrying about implementation issues and, of course, the limited number of instructions that can occur makes for short proofs. You should now be able to do Assessed Exercise 3.2, but if you have trouble with that you may want to study some of the examples for constructing Turing machines from other TMs which occur in Section 4.7.

4.4 The Halting Problem

We start our investigation of what kinds of problems are computable by asking the question of whether there are some problems which we cannot solve with a computer. In order to answer this question we will apply a trick that creates a large number of paradoxes in different disciplines, that of *self-application*.[28] The first ingredient to this is the following observation. Computer programs deal with data, which is held in the memory. Computer

[26] This seems all the more striking given the fact that Turing machines were first invented in the thirties, before anything but mechanical computers existed.

[27] And even in mathematics there are several ways of doing this—the λ-calculus being another example.

[28] If you find this sort of question interesting you may enjoy reading the by now classic 'Gödel, Escher, Bach' by Douglas R. Hofstadter.

programs themselves also are held in the memory. Hence we can consider one computer program as *the input* of another program. Programs that take programs as input are nothing new to you—you have seen parsers in the other part of the course, and you have worked with compilers when programming in C or Java.

Whenever you run a C or Java program you cannot know in advance whether or not it will terminate. Much energy has been spent on having compilers pick up as many programming mistakes as possible so as to make it easier to eliminate them, and that is how typing disciplines (such as in ML) came about. In Java, the compiler will throw an exception if you try to divide by 0. On the other hand, if you divide one variable by another, the compiler will not even try to check whether the denominator might be 0—that is something that will only be found out at runtime. You might think that it should be possible to write more sophisticated compilers which pick up on this, or you might wonder whether it isn't possible for the compiler to check for an infinite loop in a program. This is the example we investigate in detail.

Assume that we have written a program `Halt` which can decide whether or not a given other program (for a given) input loops forever, or not. In other words `Halt` tells us whether the given program *halts* after a finite number of steps (or 'terminates'), or not.[29] It might do so by printing something on the screen, or by storing the result in a boolean variable. Assume this happens in some language you are familiar with, like Java or C. That is, `Halt` expects two files as its input, the first holding a program and the second holding input data for that program. Because we do not make any stipulations that the file for the 'input data' (`file2`) actually suits the program in `file1` we would likely get a lot of error messages, but that is of no concern to us. All we are interested in is whether or not a program run with the data of that input file would terminate. So a call of `Halt` would look as follows:

```
Halt file1 file2.
```

Let us assume that `Halt file1 file2` will print 'does terminate' onto the screen if the program in `file1` terminates when run with `file2` as its input, and that it will print 'does not terminate' in the alternative case. Note that *no other case can arise*!

We now write a new program which we call `DHalt`: Take the code of `Halt`. Now do the following

- Replace every print statement of 'does terminate' by a non-terminating loop, for example

```
x=0;
while (x >=0) do
  x=x+1;
```

- Replace every print statement of 'does not terminate' by a print statement of 'does terminate'.

This is a purely syntactical operation which does not involve any programs being run. When we are finished we have a program `DHalt` which when run can behave in one of two ways: Either it loops forever (in the case where `Halt` would have terminated with printing 'does

[29] In reality such a program would typically use up all the memory available and then crash with an error message, although it might take a while until it gets there. But since we are looking at an idealized world, we will assume for the moment that the program really would run forever. You can apply an argument similar to the one given below to the outcome 'terminates with an error message'.

terminate') or it terminates and prints 'does terminate' (in the case where `Halt` would have terminated with printing 'does not terminate').

Unlike `Halt`, `DHalt` only takes one input file and uses it as *both*, the program under consideration and the input file for that program. In most cases this would, of course, lead to lots of errors, but again, that is of no concern to us here. This is the first self-application we introduce: The input program uses its own code as its input file. Hence a typical call of `DHalt` is

`DHalt file`.

We add another self application to that: We ask the question of what happens if `DHalt` runs 'on itself', that is if the input file `file` for `DHalt` holds the code for `DHalt` itself. We call this particular input file `dhalt`. Note that only two cases can arise when running `DHalt dhalt`: Either the program terminates and prints 'does terminate', or the program does not terminate.

Case 1 `DHalt dhalt` terminates.

- Because of the way `DHalt` is designed, it has to print 'does terminate'.
- By definition of `DHalt`, `Halt dhalt dhalt` prints 'does not terminate'.
- By definition of `Halt`, `DHalt dhalt` does not terminate.

This is a contradiction to the assumption that `DHalt dhalt` terminates.

Case 2 `DHalt dhalt` does not terminate.

- By definition of `DHalt`, `Halt dhalt dhalt` prints 'does terminate'.
- By definition of `Halt`, `DHalt dhalt` terminates.

This is a contradiction to the assumption that `DHalt dhalt` does not terminate.

In either case we have derived a contradiction, so a program like `DHalt` cannot exist, but that means that a program like `Halt` cannot exist either. In other words there can not be a program which decides whether a given program will terminate on a given input. Note that we have very much made use of the fact that programs can run on other programs, and in particular that programs can run on themselves. You can now try Exercise 32 and Assessed Exercise 3.3 which ask you to think about the decidability of some other questions.

This was an argument using programs in a language like `C` or `Java`, but what if we want to ask a similar question for Turing machines? We have argued earlier that all notions of computability are equivalent. But how do we apply a Turing machine to itself as input?

4.5 Turing machines and computability

At first sight, machines of any form (finite state automata, pushdown automata, Turing machines, etc) are something very different from a program. We typically think of a program as the code, that is a piece of syntax—maybe more daringly even as a word in a language. A long string of symbols, in other words. A machine, on the other hand, seems to be something more physical, an entity which does something (even if we acknowledge that these machines are theoretical devices and therefore don't live anywhere as physical entities). But how accurate is that, really?

We have seen before that we can describe the information contained in a DFA or NFA by using a regular expression—a piece of syntax. Similarly, given a non-deterministic push-down automaton we can find a grammar (another piece of syntax) which describes the same language. And looking at the formal definition of a Turing machine gives some indication that, in fact, we can describe Turing machines via some syntax as well. We can certainly code up all the states as integers (which we can then represent in binary notation if we like), which leaves the transition function. But each entry in the transition table can be coded via a quintuple

$$\langle \text{no of state} \rangle \ \langle \text{current symbol} \rangle \ \langle \text{no of new state} \rangle \ \langle \text{symbol to write} \rangle \ \langle \text{move} \rangle.$$

If we reserve a special symbol, such as #, to separate the entries of such a quintuple, and to separate different quintuples from each other, then we can write out the entire table as a long. If we want to keep track of accepting states, we may list those after we have listed the transition table, again using the symbol # as a divider. Hence we can code every Turing machine as a string over the alphabet $T \cup \{\#\}$. If we want to, we can translate every symbol in $T \cup \{\#\}$ into a binary presentation (but care has to be taken so that we can still recognize where the code for one symbol ends and that for the next symbol begins). Hence we can code every Turing machine as a binary string. Given a Turing machine M we use $\langle M \rangle$ to refer to its code (over some given alphabet). Note that not all strings over the chosen alphabet will be valid codes for a Turing machine. Also note that different transition tables (and thus different strings) can describe TMs which have precisely the same output when run on the same input. Finally note that even for the same alphabet, there may be lots of different ways of encoding Turing machines, leading to different codes for the same machine. We are not interested in *which* code is being used, just that one has been picked.

Since we can list all binary strings we can also list all Turing machines (leaving out strings which are not meaningful when interpreted as Turing machines). This allows us to assign numbers to Turing machines, and speak of the first, second, nth Turing machine—however, such numbers will depend on the encoding chosen. Hence while there are infinitely many Turing machines, there are only *countably many*.

This allows us to make a fairly crude argument why there *have* to be problems which a Turing machine (or any other computing device, if we believe in the Church-Turing Thesis) cannot decide. Let Σ be an alphabet containing at least one symbols. How many languages are there over this alphabet? A language is a subset of Σ^*, the set of all finite strings over the alphabet. So we are asking how many subsets Σ^* has. The elements of Σ^* can be listed: Start with the empty word, then list all the words consisting of one letter, then all those consisting of two letters, and so on. Therefore Σ^* is again countable. It is also an *infinite* set. But Cantor's diagonal argument tells us that the number of subsets of an infinite countable set cannot be countable.

Here is a quick summary of that argument. Let A be an infinite countable set, that is we can list its elements as $a_1, a_2, \ldots$ and that list will go on forever (but capture all elements of A eventually). We assume that this list has no repetitions. Let us assume we have a list of subsets of A, say $A_1, A_2, \ldots$. We can describe such a set in a table by saying which elements of A belong to it.

Set\element	a_1	a_2	a_3	a_4	a_5	a_6	...
A_1		✓		✓	✓		...
A_2	✓	✓					...
A_3		✓	✓	✓	✓		...
A_4			✓		✓		...
⋮							
B	✓			✓	?	?	...

For example the set A_1 contains the elements a_2, a_4 and a_5, but not the elements a_1 and a_3. We construct a new subset B of A. For that we have to decide in which column to make a tick (✓). We do that as follows. Go to the first line. The element a_1 has no tick. We make one for B. Go to the second line. The element a_2 has a tick. We do not make one for B. do not make a tick for B. In general, to decide whether there should be a tick in the ith column (that is for element a_i) we look at the ith column in the ith row and do the opposite of what we find there (no tick if there is a tick, otherwise we choose the tick). In the table above the relevant positions are framed. The result is a subset B of A with the property that it does not appear in the list: it will differ from the ith set A_i when it comes to the ith element a_i of A.

Hence we have shown that no list of subsets of A can contain *all* the subsets of A, and therefore the set of subsets of A is not countable. This method is often called the *diagonalization method* and we will meet it again. If you would like to try your hand at a proof using diagonalization yourself, try Exercise 33.

Since there are uncountably many languages over Σ, but only countably many Turing machines (over some tape alphabet that includes Σ) it is clear that there must be languages (uncountably man, in fact!) which are not Turing-recognizable.

We finally fulfil the promise given in Section 3 and describe a language which is Turing-recognizable but not Turing-decidable. Assume we have Turing machines coded into strings over some alphabet Σ, so that $\langle M \rangle$ is the code for the Turing machine M. We are now interested in a language of the following form:

$$L_{\mathrm{TM}} = \{(\langle M \rangle, \alpha) \mid M \text{ is a Turing machine which accepts } \alpha\}.$$

Theorem 4.1 *The language L_{TM} is not decidable.*[30]

Proof. See Exercise 34. □

This is the 'Turing machine version' of the Halting Problem. Just as the Church-Turing Thesis predicts, Turing machines have the same limitations as other computational paradigms.

In order to argue that L_{TM} is Turing-recognizable, let us consider the 'universal Turing machine' U. This machine can simulate every other Turing machine—in other words, given a coded description $\langle M \rangle$ of a Turing machine M, and a string α, U will behave like M when M is given the input string α. U is the recognizer for the language L_{TM} from Theorem 4.1. In order to describe how U behaves we assume that we have two tapes at our disposal, one holding the code $\langle M \rangle$ for M, and the other holding α. What U then does is to use the first

[30]Since it does not matter which notion of computation we use we will often drop the 'Turing' from 'Turing-decidable'.

tape to read off what M would do when faced with input string α, and it then mimics these actions on the second tape. The TM U accepts the pair $(\langle M\rangle, \alpha)$ if the process of simulating running M on input α terminates and M is in an accepting state at that time. (That means that if the process terminates in a non-accepting state for M, or if it does not terminate at all, then the pair $(\langle M\rangle, \alpha)$ is not accepted. Note that U *recognizes* L_{TM} but does not decide it (which would, of course, be a contradiction to Theorem 4.1.

Now for a language which is not even Turing-recognizable. Again we assume that, given a Turing machine M, we have a way of assigning a encoding $\langle M\rangle$. As we argued before, we can list all the encodings for Turing machine and use this as a list of all Turing machines, so that we may refer to the first, second, ... nth Turing machine (with respect to that encoding). There is a relation between Turing machines and codes of Turing machines, namely whether the Turing machine, say M, accepts or rejects the code $\langle M\rangle$. We can write all this down in an (infinite) table.

	$\langle M_1\rangle$	$\langle M_2\rangle$	$\langle M_3\rangle$	$\langle M_4\rangle$	$\langle M_5\rangle$	$\langle M_6\rangle$	...
M_1	accept	reject	accept	accept	reject	accept	...
M_2	reject	reject	reject	reject	reject	accept	...
M_3	reject	accept	accept	accept	accept	reject	...
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

Now consider the language[31]

$$L_d = \{\langle M_n\rangle \mid M_n \text{ does not accept } \langle M_n\rangle\}.$$

We claim that this language is not Turing-recognizable. Assume that there is a Turing machine R recognizing it. Because our list contains all Turing machines, there must be some n such that R is the nth Turing machine, that is $R = M_n$. There are two cases that can arise.

Case 1 R accepts $\langle R\rangle$, so $\langle R\rangle$ is in the language recognized by R, which is (by definition of R) L_d. But by definition of L_d, R does not accept $\langle R\rangle$. This is a contradiction.

Case 2 R does not accept $\langle R\rangle$, so $\langle R\rangle$ is not in the language recognized by R, but it is in L_d by definition of L_d. Again this is a contradiction.

Since both these cases lead to a contradiction, R cannot exist, and therefore there is no Turing machine which recognizes L_d. If you look carefully then you will notice that we have applied a diagonalization argument, just as we did when proving Cantor's result: We have ensured that our constructed machine R cannot be equal to any of the M_n. However, the contradiction we derive is different: Whereas before we had to conclude that our list could not contain all the possible subsets, in this example we know that all Turing machines occur in the list, so the conclusion must be instead that a Turing machine like the one we just constructed cannot exist.

More examples for properties to look at can be found in Exercises 35 and 36.

4.6 Some decidable problems

We have already seen in Exercise 24 that given a DFA A we can define a Turing machine D which mimics the action of A. But we can do better than that: Just as we can have a Turing

[31] The 'd' stands for 'diagonal'. Can you see why?

machine which simulates every other Turing machine we can have a Turing machine which simulates every DFA (that is easier, in fact). Hence the following language is decidable

$$\{(A, \alpha) \mid A \text{ is a DFA that accepts the string } \alpha\}.$$

Since DFAs and NFAs have the same power, this also means that the language

$$\{(A, \alpha) \mid A \text{ is an NFA that accepts the string } \alpha\}$$

is decidable, and similarly for

$$\{(R, \alpha) \mid R \text{ is a regular expression which is matched by the string } \alpha\}.$$

More interestingly, we can decide whether a given DFA recognizess the empty language, and we can decide whether the languages recognized by two DFAs are equal.

For context-free languages, Proposition 3.1 can be used to show that the language

$$\{(A, \alpha) \mid A \text{ is a PDA that recognizes the string } \alpha\}$$

is decidable. The problem of whether a given PDA recognizes the empty language is decidable, but that whether two PDAs recognize the same language is not!

4.7 Some undecidable problems

We have already seen some undecidable problems, namely the Halting Problem or the decidability of L_{TM}. Many proofs of undecidability of other problems proceed by showing that if the new problem were decidable then so would some old problem be.

A relatively simple example is the language

$$L_{\text{halt}} = \{(\langle M \rangle, \alpha) \mid \text{the Turing machine } M \text{ halts on input } \alpha\}.$$

This is the Halting Problem for Turing machines, and if you have done Exercise 34 you have seen a direct proof that it is not decidable. Here is another proof. Assume we have a TM H which decides L_{halt}. We build a new Turing machine D as follows. Given the coding of a Turing machine $\langle M \rangle$ and an input string α, run H on $\langle M \rangle$ and α. H will have to halt. If H accepts, run M on α, which will have to halt, and behave like M to accept or reject α. If H rejects, reject. The new TM D decides the language L_{TM} which is undecidable by Theorem 4.1.

This illustrates the principle of *reduction*. We have shown that if L_{halt} is decidable then so is L_{TM}, that is, we have *reduced* the question of the decidability of L_{TM} to that of the decidability of L_{halt}.[32]

Here is another example. Consider the language

$$L_{\text{empty}} = \{\langle M \rangle \mid \text{the language recognized by the TM } M \text{ is the empty language}\}.$$

[32]The 'reduce to' relationship might be the opposite way round from what you expected! This is because the thought that counts here is not 'we have reduced a new problem to one which has been solved already', but 'given a solution to the new problem we can find a solution to the old problem', which shows that anything solving the new problem is more powerful (well, at least as powerful) as something that solves the old problem. So solving the new problem *entails* solving the old problem which we know cannot be solved. As a consequence we know that the new problem cannot be solvable either.

This language is not decidable for the following reason. Assume there is a Turing machine D which decides it. Again we want to reduce the problem of the decidability of L_{TM} to this new problem. Using D we build a machine E which decides L_{TM} as follows.

This TM E will have to deal with inputs $(\langle M\rangle, \alpha)$. Given a code for a Turing machine M we define a changed Turing machine M' which behaves as follows: Given some input other then α, M' rejects. Given the input α, M' simulates M and accepts or rejects just as M does. Clearly the language recognized by M' is either empty (if M does not accept α) or consists of the one word α (if M accepts α).

In order to define the TM E we describe its behaviour on input $(\langle M\rangle, \alpha)$. First it simulates running D on the input M'. If that process terminates in an accepting state then we know that M rejects α, so M' recognizes the empty language. In this case we let E reject, and otherwise we let E accept. But now the machine E decides L_{TM}. This cannot be, so E cannot exist, which means that D cannot exist either and therefore L_{empty} is undecidable.

You are asked to apply this technique in Exercise 37.

In some sense all the really interesting questions which one might ask about programs are undecidable. All these are about the *behaviour* of a program rather than its *syntactic* properties.

Theorem 4.2 (Rice's Theorem) *Let P be a property such that there are some Turing-recognizable languages which satisfy P, and some Turing recognizable languages which do not satisfy P.*[33] *Then P is undecidable.*

Proof. We reduce the Halting Problem to this new problem. Since P is non-trivial there exists a Turing-recognizable language L satisfying P. Let R be the TM which recognizes L.

First case: the empty language does not satisfy P. Given an input $(\langle M\rangle, \alpha)$ for the Halting Problem proceed as follows.

Define a new machine M' that on input β does the following: It saves β on some separate tape and writes α on another tape. (Note that M' depends on α.) It then simulates running M on the input α. If M halts on α it runs R on β and accepts if R accepts.

Two cases can arise. The first case is that M does not halt on α which means that the machine M' never gets beyond simulating running M on α, and therefore M' will not accept the input β. Since this will be true for every β, the language recognized by M' is empty in this case. The second case is that M does halt on α and the machine proceeds to running R on β, then M' accepts β if and only if R accepts β. This is the case if and only if $\beta \in L$.

In summary, M halts on α if and only if the language recognized by M' is L if and only if the language recognized by M' satisfies P. (This is where we use that the empty language does not satisfy P.) We have thus reduced the Halting Problem to the set

$$\{\langle M'\rangle \mid \text{the language recognized by the TM } M' \text{ satisfies } P\}.$$

Therefore this problem is not decidable.

Second case: the empty language does satisfy P. We consider the opposite property $\overline{P}$ instead, and now are interested in the set of languages not satisfying P (this set will not include the empty language). Applying what we have just shown for $\overline{P}$ in the place of P we know that the set of languages not satisfying P is undecidable. If we assume that P is

[33] In such a case we also say that P is *non-trivial*. Trivial properties are boring, because they are satisfied either by all languages or by none.

decidable then so would its complement be by Theorem 3.2. Since this gives a contradiction, the set of languages satisfying P must be undecidable as well. □

Further properties are examined in Exercise 38.

4.8 Applications in mathematics

We have seen above that there are some limitations to what we can ask programs to do. In particular we can never have a program which is capable of checking whether every program terminates. That does not mean that it would not be possible to write programs which do this for *some* programs, for example. Nor does it mean that we can never hope to check *individual programs*—all this says is that there is no algorithm which works for all programs.

This is a limitation which does not just concern computers and computer science, it has applications which you might not have anticipated for areas such as mathematics. For a long time mathematicians believed that all questions could be solved *in principle*, for example that a given (well-formed) statement about, say, arithmetic, is either provably true or provably false, and that problems involving integers should be solvable using algorithms.

One such problem, stated by the German mathematician David Hilbert in 1900, was that of finding an algorithm that, given some polynomial (possibly in several variables), would decide whether there are integers that can be substituted for the variables so that the polynomial evaluates to 0.[34] Only in 1970 was it shown that such an algorithm cannot exist. To give a negative answer to this problem, a formal notion of algorithm had to be developed first, together with a notion of decidability. The language of polynomials that 'have integer roots' is another language which is Turing-recognizable but not Turing-decidable.

Here is another interesting consequence that a program which solves the Halting Problem would have. Take the famous equation

$$a^n + b^n = c^n,$$

where a, b, c, and n are all integers. The French mathematician Fermat conjectured in the 17th century that this equation has no solutions if n is greater than or equal to 3. For more than 300 years mathematician tried to prove this [35] and only in the last decade did the British mathematician Andrew Wiles manage to do so.[36] But if we had a program `Halt` solving the Halting Problem, we would have known much earlier whether this conjecture was indeed true.

How would we do that? Well, it is relatively easy to write a program `FSearch` which searches for solutions to the problem by systematically testing all possible values for a, b, c and n. We would then take our program `Halt` and run it on the program `FSearch`. If `FSearch` terminates, then there must be a solution (and we can run `FSearch` for a finite(!) time to find them). If it does not then we know there are no solutions. Thus considerations about the Halting Problem also tell us something about an approach to having mathematics done by computers.[37]

[34] This is the tenth of the twenty-three problems Hilbert identified as a challenge for the coming century when addressing the International Congress of Mathematics.

[35] This is also known as 'Fermat's Last Theorem'.

[36] His proof incorporates so much of algebraic geometry, in particular the theory of elliptic curves, that we know it cannot possibly have been anything that Fermat himself might have done. The full proof takes up some 200 pages, although people are working on simplifying it.

[37] Note, however, that based on this we cannot rule out that there may not be other ways of solving math-

An interesting undecidable problem from mathematics is the language of all first order propositions, that is all statements in first order logic which are provable. This means that there cannot be a 'perfect theorem prover'!

Summary. In this section we have seen that Turing machines can be considered as very general computation devices. Further Turing machines are very robust against making changes to their memory device (as long as it is infinite), and non-deterministic Turing machines are precisely as powerful as deterministic ones. The Church-Turing Thesis says that they are precisely as powerful as any other computation device. It therefore makes sense to split problems into 'decidable' and 'undecidable' ones—those which can be solved by such a device and those which cannot. The most famous among the latter is the so-called Halting Problem which asks whether a program (or Turing machine) will halt on a given input. Because there are undecidable problems there are questions which computers can never solve. This is the reason why compilers only deal with syntactic properties of programs. All non-trivial properties about Turing-recognizable languages are undecidable.

---

ematical problems using computers in a systematic manner—it is just the question of termination which we cannot utilize.

4.9 Exercises for Section 4

Exercise 27 (a) Design a TM which finds the first a of the input string.

(b) Design a TM which find the first stretch of two consecutive as of the input string.

(c) *Assume that somebody placed the head of a TM somewhere other than at the first symbol of the input string. Design a transition function which finds the input string (you may assume that the tape is otherwise empty), no matter where the head is when the computation starts. (Hint: You may want to add a new special symbol to the tape alphabet T to help with this.)*

Exercise 28 Do the following by describing how your machine is supposed to work similar to the instructions 1. to 5. in the example of the 'copy' TM. There is no need to give the actual transition table. You may make use of machines that have been introduced so far if you like.

(a) Design a TM which prints one a, followed by a blank, followed by two as, followed by a blank, followed by three as, and so on, forever.

(b) Design a TM which decides whether two strings on the tape, separated by a blank, are equal. (The machine is allowed to 'destroy' the words in the course of the computation.)

(c) Design a TM which adds 1 to a string of 0s and 1s on the tape, where we assume the string describes a number in the usual binary notation.

Exercise 29 *Assume you have a Turing machine with a two-dimensional tape. How would you go about assigning cells on a one-dimensional tape to the cells on the two-dimensional tape so as to keep the same information? (Note that it is not sufficient to try to 'string all the cells on the same row together before all the cells on the next row'.)*

Exercise 30 *Show that a Turing machine can be simulated by a PDA with* two *stacks. Think about how the content of the tape is used. (Hint: Use one stack to store the content of the tape to the left of the head, and the other for the content of the tape to the right of the head.)*

Exercise 31 (a) *Assume you have a machine which has two counters. In one step these counters can be incremented or decremented by one (independently from each other) and there is a test of when the counter has reached* 0. *With such a machine, how would you carry out the following tasks? (If you find this description too vague, maybe you prefer the following: Again we have a finite set of states. The transition function maps a triple consisting of a state and two truth values (encoding the test for 0 for the first/second counter) to a new state and two actions (to be carried out on the two counters) which are elements of* {decrease, increase, do nothing}.*)*

1. *Double a number (that is, double the contents of one of the counters).*
2. *Divide a number by* 2 *(integer division, this ignores any remainder).*
3. *Find the remainder of a number when divided by* 2.
4. *Can you generalize your algorithm from 3.) to apply to division by any number? For this one you may assume that we have three counters. If you know in advance which number you are dividing by, can you make do with two counters? How about multiplication?*

(b) *Suppose you are given a machine with three counters instead. Is it more powerful than the one with two counters? (Hint: The content of all three counters can be held in just one using the following coding trick: If the contents of the counters are, say, a, b, and c, then the number $2^a 3^b 5^c$ allows us recapture a, b, and c.) Argue that adding any number of counters will not make this machine more powerful. (There is no need to do a formal proof here, a careful argument will do.)*

(c)* *Show that you can simulate the two-stack machine from Exercise 29 using a two-counter machine. Hence two-counter machines are as powerful as Turing machines. (This is not all that difficult but requires some careful thinking. How do you code the symbols on the stack? Hint: Assign a prime number to each of them. Then use a counter for each stack and decide how to simulate popping from/pushing onto the stack using counters.)*

Exercise 32 (a) Is it possible to write a program which, when given a program and an input file, will check whether that program will run for more than 1000 machine cycles on that input file?

(b) Is it possible to write a program which will check whether a given piece of code in some other program will ever by executed? (This is also called the problem of *dead code*.)

Exercise 33 (*) *Use the diagonalization method to show that there are uncountably many real numbers in the interval* $[0, 1]$. *(Hint: Use the decimal notation to construct a diagonalization argument.)*

Exercise 34 *Prove Theorem 4.1. Do so by mimicking the proof that no program can solve the Halting Problem. (Hint: First assume that the language is decidable. This means that a certain Turing machine must exist. Now construct a new Turing machine based on the first one and apply that machine to itself.)*

Exercise 35 (a) *Show that the language*

$$L_{\textit{non-empty}} = \{\langle M \rangle \mid \textit{the language recognized by } M \textit{ is non-empty}\}$$

is Turing-recognizable.

(b) *Show that a language is decidable if and only if it as well as its complement are Turing-recognizable. (Hint: Remember Theorem 3.2.)*

Exercise 36 (a) Is the following true or not? Given two languages L and L' over the same alphabet Σ such that $L \subseteq L'$. If L' is decidable, then so is L. Argue your case!

(b) Show that the union of two decidable languages is decidable.

(c) *What about the union of two Turing-recognizable languages? How do you have to change your argument to the fact that recognizers do not have to halt? (Hint: Recall that given two TMs we can build a new one which runs both of these 'in parallel'.)*

(d)* *Consider the other operations on languages introduced in Section 1, such as intersection, complement, reversal, concatenation, Kleene star. If you apply these to Turing-decidable (Turing-recognizable) languages, do you get a Turing-decidable (Turing-recognizable) language?*

Exercise 37 (a) *Show that the language*

$$\{(\langle M\rangle, \langle M'\rangle) \mid \textit{the TMs } M \textit{ and } M' \textit{ recognize the same language}\}$$

is undecidable. (Hint: Reduce this to the decidability of L_{empty}*.)*

(b) Show that the language L_{empty} is not Turing-recognizable. (Hint: Use Exercise 35.)

Exercise 38 (a) Show that the question of whether two PDAs recognize the same language is not decidable.

(b) Argue that the question of whether the language of a Turing machine is infinite is undecidable. *Is it Turing-recognizable? Just give a short argument, do not try anything more formal.*

(c) *Show that the set of codes for Turing machines which, when started on an empty tape, will eventually print a non-blank symbol is decidable. (Hint: Argue that a Turing machine which does not stop will eventually enter a loop and make use of that.)*