Turing machines and computable functions[*]

One of the main outcomes of the course is to understand the difficulty of a certain task. In particular, we study the complexity associated to the decision problem of languages. In increasing order of complexity, we consider: regular, polynomial time computable, nondeterministically polynomial time computable, computable in polynomial space, decidable, and recognizable. The previous chapter focussed on the least complex languages: the regular languages. In the next chapter we focus on the 2 classes on the other extreme: the decidable and recognizable languages. For this we need the concept of a *computable function*. We address the question: "What is the largest class of functions that are somehow computable by a discrete mechanical process?"

Probably, most of you are familiar with a few programming languages. Nowadays, it is easy to believe that in your favorite programming language, any discrete mechanical process can be simulated. Unfortunately, the description of your favourite programming language can not be used as a mathematical definition: its full specification is too long, and someone else might insist on using his own favourite language (there exist teachers who use λ-calculus or even self-invented languages). Still, we want a powerful enough definition from which we can easily believe that our favorite programming languages can be "implemented" with it.

To prove that a problem can not be solved by an algorithm, we typically show that a solution for it is as hard as evaluating all computable functions. The idea is to construct instances of the problem that somehow "implement the same features" as used in our definition of computable functions. For this reason we want a definition that is as simple as possible. In summary, we need a definition that is simple and powerful enough.

Turing machines present a solution. They are simple to define, and we can implement some simple programming language with them. After defining computable functions using such machines, we argue that the definition is robust: having more tapes, increasing the tape alphabet, changing the possible movements, does not change the corresponding class of computable functions. Then we study the speed with which they compute. In particular we show that this depends on the number of tapes it has, but this dependence is polynomial. Afterwards, we argue that Turing machine can implement features of typical programming languages. Finally, we show that there exist Turing machines that can simulate all other machines.

Only in this lecture we will talk about low level details of algorithms. The hope is that after this lecture you are convinced that (1) Turing machines are precise mathematical objects,

[*]Theoretical Computer Science 2017-2018, National Research University Higher School of Economics.
First version: February 10, 2018.
Update: February 16, 2018, moved solutions to the end, correcting typos,
a small change in the definition of multitape machines.
Update: February 17, 2018, added exercise 9, 14–19, and solutions.
Update: February 28, 2018, corrected typo in the Halting condition for Turing machines.

(2) they can model any type of discrete computation, and (3) that you are able to estimate the time and space they needed to complete some tasks.

1 Definition

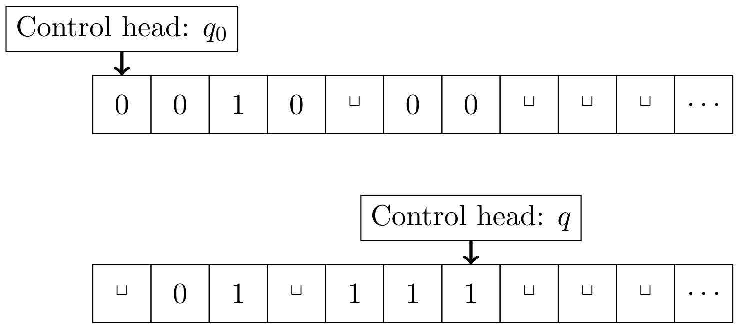


Figure 1: The initial configuration for the computation $U(0010, 00)$ and a configuration in the middle of some computation.

A Turing machine is a computing device that on input one or more strings might produce as output a string. The machine has two parts: a tape and a control unit (also called computation head). The control unit is similar to a finite state machine and is characterized by a finite set Q of *control states*, a *start state* $q_0 \in Q$ and a *transition function* d. The tape is a one-way infinite sequence of cells and each cell contains a symbol from some tape alphabet Γ. In Γ there is a special symbol $\sqcup$, called the *blank*. The control unit is always located above a cell of the tape and it can move left or right.

A computation is performed as follows. Initially the control unit is located at the leftmost cell of the tape and it is in a special control state q_0, called *start state*. All cells of the tape are blank except at the beginning of the tape, where the input strings are written, separated by blanks, see figure 1. During the computation, the control unit transforms the tape step by step. At each step, the control unit has a unique control state. Depending on this state and the symbol below the control head, the transition function d, specifies

- which symbol is written in the cell below the computation head,
- whether the computation head moves one cell left or right,
- the new state of the computation head.

The transition function is a partial function. If it is undefined for some state and symbol, or the head moves left on the first cell of the tape, the computation is finished. If this happens, and the tape contains $r\sqcup^\infty$ for some string r that has no blanks, then r is the output of the computation. For convenience, we require that the computation head returns to the left, and either halts on the first cell or falls of the tape.

We define the procedure above formally. The transition function is a function

$$d : Q \times \Gamma \to \Gamma \times \{-1, 1\} \times Q.$$

A value $d(q,a) = (b,m,r)$ means that if the control head is in a state q above a cell containing $a \in \Gamma$, it writes $b \in \Gamma$, moves one cell left ($m = -1$) or right ($m = 1$), and finally, changes to the control state $r \in Q$. A computation is just a repetition of such transformations.

Notation. For a finite set A, let A^∞ be the set of infinite sequences of elements in A. The ith element of such a $T \in A^\infty$ is written as $T_i \in A$. Let $a^\infty = aa\ldots$ Let $\sqcup$ be a fixed symbol. Let $\mathbb{N} = \{1, 2, 3, \ldots\}$.

At each computation step, the state of the Turing machine is described by a triple $(i, q, T) \in \mathbb{Z}_{\geq 0} \times Q \times \Gamma^\infty$ where i is the position of the computation head ($i = 0$ indicates that the head has fallen of the tape), $q \in Q$ is its control state and $T \in \Gamma^\infty$ the contents of the tape. These triples define a directed graph where an edge from one state to another represents a transformation that can happen in 1 computation step.

Definition 1. *A* deterministic Turing machine *is a* 4*-tuple* (Q, Γ, d, q_0) *where* Q *and* Γ *are finite sets,* $q_0 \in Q$*,* $\sqcup \in \Gamma$*, and* $d \colon Q \times \Gamma \to \Gamma \times \{-1, 1\} \times Q$ *is a partial function. Consider the directed graph whose vertices are elements of* $\mathbb{Z}_{\geq 0} \times Q \times \Gamma^\infty$ *(called* states*). There is an edge from a state* (i, q, T) *with* $i \geq 1$ *to the state*

$$(i + m, \quad r, \quad T_1 \ldots T_{i-1} b T_{i+1} \ldots)$$

if $d(q, T_i) = (b, m, r)$*. There are no other edges.*

If for $r, u, v, \ldots, z \in (\Gamma \setminus \{\sqcup\})^*$ *the unique path starting in the state* $(1, q_0, u \sqcup v \sqcup \ldots \sqcup z \sqcup^\infty)$ *is finite and ends in* $(0, q, r\sqcup^\infty)$ *or* $(1, q, r\sqcup^\infty)$ *for some* $q \in Q$*, we say that* U outputs r *on input* $u, v, \ldots, z$ *and write* $U(u, v, \ldots, z) = r$*, otherwise the value is undefined.*

Note that a machine halts if either the transition function is undefined or if the computation head falls of the tape, i.e., reaches position $i + m = 0$. Note that there are no outgoing edges in states with position 0. There exists several variants of the definition and here one is given that is simplified for our purposes. Note that in order to produce some valid output, we require that the machine cleans up its tape. This requirement is convenient for induction purposes. We also use Turing machines that can decide a language.

Definition 2. *A Turing machine* (Q, Γ, d, q_0) decides *a language* L *over* Σ *if there exists a set* $F \subseteq Q$ *such that (1) for all* $x \in \Sigma^*$ *the path starting in* $(1, q_0, x\sqcup^\infty)$ *is finite, and (2) this path ends in a state from* $\mathbb{Z}_{\geq 0} \times F \times \Gamma^\infty$ *if and only if* $x \in L$*.*

A language is computable *or* decidable *if there exists a Turing machine that decides it.*

When deciding a language, we do not care about the final position of the computation head and the contents of the tape after the machine halts. In the part of computational complexity we use the nondeterministic variant of such machines. We present the definition here for the sake of completeness.

Definition 3. *A* nondeterministic Turing machine *is a* 4*-tuple* (Q, Γ, D, q_0) *where* Q *and* Γ *are finite sets,* $q_0 \in Q$*,* $\sqcup \in \Gamma$*, and* $D \subseteq Q \times \Gamma \times \Gamma \times \{-1, 1\} \times Q$*. Consider the graph with vertices* $\mathbb{Z}_{\geq 0} \times Q \times \Gamma^\infty$ *and directed edges from* (i, q, T) *to* (j, r, S) *if* $i \geq 1$*,* $T_k = S_k$ *for all* $k \neq i$*, and*

$$(q, T_i, S_i, j - i, r) \in D.$$

The machine recognizes *a language* L *over* Σ *if there exists an* $F \subseteq Q$ *such that for all* $x \in \Sigma^*$*:* $x \in L$ *if and only if there exists a path in this graph that starts in* $(1, q_0, x\sqcup^\infty)$ *and ends in an element from* $\mathbb{Z}_{\geq 0} \times F \times \Gamma^\infty$*.*

2 Examples

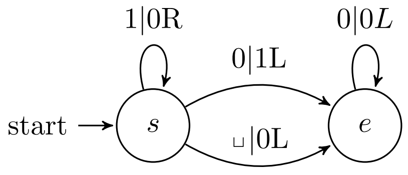
We give some detailed examples of Definition 1. These examples will be used in the construction of a universal Turing machine.

An enumerator of binary strings. Let ε be the empty string. We construct a machine U such that the series $\varepsilon, U(\varepsilon), U(U(\varepsilon)), \ldots$ contains all binary strings exactly once. The following machine maps every string to the next string in reversed lexicographic order:

$$\varepsilon, 0, 1, 00, 10, 01, 11, 000, 100, \ldots$$

Let $L = -1$ and $R = 1$. The machine is given by $(\{s, e\}, \{0, 1, \sqcup\}, d, s)$ where d is defined by

q, a	$d(q, a)$
$s, \sqcup$	$0, L, e$
$s, 0$	$1, L, e$
$s, 1$	$0, R, s$
$e, 0$	$0, L, e$



In the tables below, the subsequent states of the machine are given when started on input 111.

i	q	T					
1	s	$\underline{1}$	1	1	$\sqcup$	$\sqcup$	$\ldots$
2	s	0	$\underline{1}$	1	$\sqcup$	$\sqcup$	$\ldots$
3	s	0	0	$\underline{1}$	$\sqcup$	$\sqcup$	$\ldots$
4	s	0	0	0	$\underline{\sqcup}$	$\sqcup$	$\ldots$
3	e	0	0	$\underline{0}$	0	$\sqcup$	$\ldots$
2	e	0	$\underline{0}$	0	0	$\sqcup$	$\ldots$
1	e	$\underline{0}$	0	0	0	$\sqcup$	$\ldots$
0	e	0	0	0	0	$\sqcup$	$\ldots$

(s,1)	1	1	$\sqcup$	$\sqcup$	$\ldots$
0	(s,1)	1	$\sqcup$	$\sqcup$	$\ldots$
0	0	(s,1)	$\sqcup$	$\sqcup$	$\ldots$
0	0	0	$(s, \sqcup)$	$\sqcup$	$\ldots$
0	0	(e,0)	0	$\sqcup$	$\ldots$
0	(e,0)	0	0	$\sqcup$	$\ldots$
(e,0)	0	0	0	$\sqcup$	$\ldots$
0	0	0	0	$\sqcup$	$\ldots$

The state $(0, e, 0000\sqcup^\infty)$ has no outgoing edge, because of the condition $i \geq 1$ for the edges of the computation graph in Definition 1. Hence, the computation halts with the desired output 0000. On the right, a different representation of the computation path is given. This table with cells in $\Gamma \cup (\Gamma \times Q)$ is called a *tableau.* Tableaus are important in the study of NP-hard problems. In the next chapter we also use them to study tilings of the plane.

Exercise 1. Give a schematic representation of a Turing machine that maps any bitstring x to $0x$.

Exercise 2. Design a Turing machine that inverts the machine U constructed above; more precisely, maps ε to ε and every other string to the preceeding one in $\varepsilon, 0, 1, 00, 10, 11, 000, 100, \ldots$ Thus, $U(0) = \varepsilon$, $U(1) = 0$, $U(00) = 1$, etc.

3 A robust definition of computable functions

Definition 4. *A partial function $f\colon \Sigma^* \to \Sigma^*$ is* partial computable *if there exists a deterministic Turing machine U such that U and f are defined for the same $x \in \Sigma^*$ and if defined, then $f(x) = U(x)$. A function is* computable *if it is total and partial computable.*

In a similar way we define (partial) computable functions with more arguments. These definitions do not change if we change our model of Turing machines: we could equivalently use two-way infinite tapes, multiple tapes (see further), and even two-dimensional tapes. The class of computable functions on Σ^* also does not change if we change the tape alphabet to $\Gamma = \Sigma \cup \{\llcorner\!\lrcorner\}$. We can also consider machines that in each computation step either writes or moves. However, some changes can be dangerous: if the computation head can not move left, the tape will become practically useless.

Recall that in our definition, a machine should always move its computation head 1 step left or right in a computation step.

Lemma 5. *The class of computable functions does not change if we consider machines that in each computation step can move left, right, or remain above the same cell.*

Proof. Assume a machine makes the following 2 computation steps when it is in control state q: it reads a symbol a, writes b, *does not move*, and goes to state r. In the next step, it reads the previously written symbol b, writes c, moves in direction $M \in \{L, N, R\}$, where N represents *no move*, and obtains control state t. These two steps can be combined in one: it could read a, write c, move in direction M and arrive in state t.

Now we know how to transform the schematic representation of the machine, to remove edges where the machine does not move. For each path of length 2 whose edges are consecutively labelled by $(a|bN)$ and $(b|cM)$ for $a, b, c \in \Gamma$ and $M \in \{L, N, R\}$ we do the following: we delete the first edge with label $(a|bN)$, and add an edge labelled by $(a|cM)$ from the start to the endpoint of the path. After this, we removed all edges in which the machine does not move, except for those after which the machine halts.

Now assume the machine is in a state q, reads a, writes b, does not move, and goes to state r in which it halts. Thus the transition function is undefined in (r, b). We can not remove the edge $a|bN$ from the schematic representation, because this changes the output. In stead we let the machine go forward to write b and go backward. For this, we add two states h_1 and h_2. For every $e \in \Gamma$, there is an edge labelled by $e|eL$ from h_1 to h_2. For every remaining edge $a|bN$, we add an edge labelled by $a|bR$ that leaves from the same state and goes to h_1. Now we have removed all edges in which the control head does not move, without changing the output of the machine. $\square$

Lemma 6. *The definition of partial computable functions does not change if we consider only Turing machines that never fall of the tape, in other words, for all inputs in Σ^*, the computation path terminates in some state (i, q, T) with $i \geq 1$.*

Proof. The idea is to simulate the original machine with the following modifications. We keep a "mark" on the first cell and if the original machine goes left on a marked cell, we let our machine halt. Formally, this is realized by extending the tape alphabet Γ with a copy $\mathring{a}$ of each symbol a.

By Lemma 5 we can construct a machine that on some computation steps does not move its computation head. The new machine first replaces the symbol in the first cell by its copy and does not move. For this we add a new start state to the set of states. After this, the machine arrives in the start state of the original machine. Then the computation of the original machine is ran with the following modifications:

- during the computation it keeps marked cells marked and unmarked cells unmarked,
- if it wants to go left or halts on a marked symbol, it writes the corresponding unmarked symbol and halts, (thus the machine does not move in this step),

By construction the first cell is always marked and the machine never goes left on this cell. The machine also computes the same partial function. □

Exercise 3. Show that if f and g are computable then also the function $t \mapsto f(g(t))$ is computable.

Decidable languages can also be defined using computable functions. Let

$$\mathbb{1}_L(x) \;=\; \begin{cases} 1 & \text{if } x \in L, \\ 0 & \text{otherwise.} \end{cases}$$

Lemma 7. *A language $L \subseteq \Sigma^*$ is decidable if and only if $\mathbb{1}_L$ is computable.*

The proof is not so hard, there are just a some technical details, for example, one needs to explain that the tape of the decider can be cleaned up, that one can return to the initial position while remembering the result and then write the result.

Exercise 4. Show that regular languages are decidable. (Note that not all decidable languages are regular, see further.)

Exercise 5. What is the set of languages decided by Turing machines that in each computation step can move their computation head only to the right?

4 Multitape Turing machines

A k-tape Turing machine is a Turing machine with k tapes, numbered from 1 to k. Each tape has its own head for reading and writing. All these heads are controlled by a single control unit. Initially all input is written on the first tape and all other tapes only contain blanks. In each computation step, all heads simultaneously read the cell, then they all write an individual symbol, and all move in an individual direction. Afterwards, the control unit transits to its new state. For convenience, we allow that computation heads do not move in a computation step. The control unit of a k-tape TM is described by a partial function

$$d\colon\; Q \times \Gamma^k \quad\to\quad \Gamma^k \times \{-1, 0, 1\}^k \times Q.$$

The computation terminates if one of the read-write heads goes of the tape, or if d is undefined for the state and the scanned symbols. The output is defined in a similar way on the first tape. We do not repeat the formal definition because it is similar; we just mention that the full state of the machine is described by a triple in $\mathbb{Z}^k_{\geq 0} \times Q \times (\Gamma^k)^\infty$.

4.1 Multitape Turing machines compute the same functions

Theorem 8. *A partial function can be computed by a multitape Turing machine if and only if it can be computed by a single tape Turing machine.*

Proof. Every single-tape machine is a multitape machine, so one direction is obvious. For each multitape TM M we need to construct a single tape TM S that simulates M.

The scanned contents of the k tapes are written after each other, separated by some symbol # that is not in the alphabet of M, see figure 2. S's tape alphabet consists of this symbol # and two copies of M's alphabet, one copy we denote with dotted symbols. These

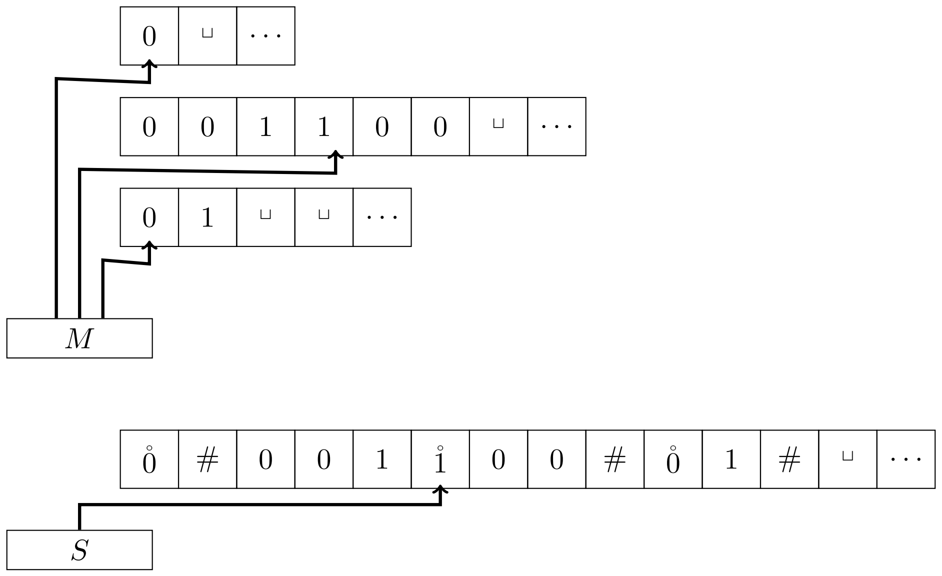


Figure 2: Definition of a multitape Turing Machine.

dotted symbols represent marked cells and are used for cells above which the reading/writing head is located.

On input $w = u{\sqcup}v{\sqcup}\dots{\sqcup}z$, S proceeds as follows:

1. First the input is transformed to the form

$$\#\mathring{w}_1 w_2 \dots w_{|w|}\#\mathring{\sqcup}\#\mathring{\sqcup}\dots\#\mathring{\sqcup}\#.$$

   After the transformation, the head returns to the beginning of the tape.

2. To simulate a single move in M, all dotted letters are scanned and "remembered" by the computation head. Then the computation head returns to the beginning of the tape.

3. S makes a second pass through the dotted letters and updates them according to the writing and the movements of M's heads. Then it returns to the beginning of the tape.

4. In the previous step, if at some point S wants to overwrite a symbol #, (this means that M performs a right move to a previously unscanned cell), then S moves the remaining tape one cell to the right, returns, and inserts the appropriate symbol.

5. If M does not halt, S repeats step 2. Otherwise, S deletes all symbols that represent the other tapes, returns left, and halts.

Each time S executes steps $2-4$ above, it simulates 1 step in the computation of M. This is repeated until M terminates, and then S also terminates. The last step guarantees that the output is the same. It can be verified that all these steps can indeed be carried out by a Turing machine. This implies the theorem. □

4.2 Multitape machines are faster than single tape machines

The proof of Theorem 8 implies a stronger result: one can simulate a k-tape machine U by a single tape machine that runs in time $O\left((\text{time}_U(x) + |x|)^2\right)$, where the implicit constants in the $O(\cdot)$ notation depend on k (but not on x). We show that this quadratic increase is necessary.

For this we study the language of *palindromes*. Palindromes are also used for words in natural languages: they are the words that remain the same when spelled backwards. In English one has the following examples: level, kayak, madam, radar, rotator. If punctuations and spaces are ignored: "test set", "race car", "Was it a car or a cat I saw?" and "Dammit, I'm mad!".

Fix some alphabet Σ. Let x^R be the reverse of a string x, i.e., $x^R = x_{|x|}x_{|x|-1}\dots x_1$.

$$\text{PAL}_\Sigma = \{x \in \Sigma^* : x = x^R\}$$

Imagine you are given a long sentence and you have to check whether it is a palindrome. How would you proceed? Perhaps the easiest way is to compare the first letter to the last letter, mark both letters, then compare the first unmarked letter against the last unmarked letter, mark them, and so on. We can speed this up by remembering several letters at a time, but this speed up is limited by the numbers we can simultaneously remember.

This strategy can be executed on a single tape Turing machine. It requires quadratic time in the length of the string, because the machine must make a linear number of comparisons and on average these comparisons are at a distance of half of the length of the string. Note that the machine can speed up by comparing chunks at once. But for large strings, the computation time remains quadratic: by definition, the control head has finitely many states, and therefore can "remember" at most finitely many symbols. We now show that this strategy is essentially optimal.

Theorem 9. *For any single tape machine U that decides* PAL*, there exists a c and infinitely many $x \in$* PAL *such that the computation time of $U(x)$ exceeds $|x|^2/c$.*

Exercise 6. Show that the language PAL can be decided in time $O(n)$ on a 2-tape Turing machine.

In the proof of the theorem, we use crossing sequences and some results about crossing sequences. The idea of our proof is to express that the machine should cross the cells in the middle many times. The sum of the lengths of these crossing sequences lower bounds the computation time. Fix a machine U.

Definition 10. *The* crossing sequence $c_i(x)$ *for a string x on position i is the list $q_1, q_2, \dots, , q_e$ of consecutive control states that U has at the end of the computation steps in which the head moves to the ith tape cell.*

See figure 3. Let $|c_i(x)|$ represent the length of the crossing sequence. Note that $\sum_{i=1}^{\infty} |c_i(x)|$ is equal to the computation time of U on input x.

Lemma 11. *If $c_i(u) = c_j(v)$, $u_i = v_j$, and U accepts both u and v, then U also accepts*

$$u_1 \dots u_i v_{j+1} \dots v_{|v|}.$$

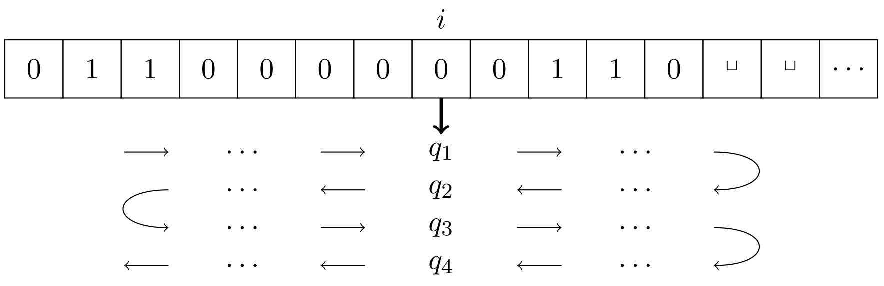


Figure 3: Crossing sequence $c_i(x)$

Proof. Consider the following communication problem between Alice and Bob. Alice has a part $w_1 \dots w_i$ of a string w and Bob has a part $w_i w_{i+1} \dots w_{|w|}$. Together they want to figure out whether $U(w) = 1$.

They can do this by running the machine on their parts of the tape, and sending the state each time the machine arrives at the border of their part.

More precisely, Alice starts to run the machine and when it arrives on the ith cell, she sends the control state to Bob. If the machine returns to her part she continues to compute, and otherwise, she waits while Bob simulates the machine on his part. If in Bob's simulation the machine arrives at the ith cell, he informs Alice of its control state. If the machine returns to his part, he continues computing, otherwise, he waits for Alice's message, and so on.

Note that at each point, both parties know who might send the next message, because each time the machine arrives in the ith cell, they both know its state, and hence the new value of the cell and the part to which it goes.

If the string $u_1 \dots u_i v_{j+1} \dots v_{|v|}$ is splitted at position i, Alice will send the same messages as in the computation for u when splitted in positions i, and Bob will send the same message as in the computation for v splitted in j. This is because the respective computations only depend on the computations on their parts and the messages previously received from the other. This follows by induction. We explain this in detail.

The first message of Alice will be $(c_i(u))_1$, because changing the cells at the right of position i does not influence her first run of computations. At some point, the machine might go to Bob's side. Bob's computation only depends on the message(s) of Alice and his part of the input. Thus both for v and $u_1 \dots u_i v_{j+1} \dots v_{|v|}$, he receives the same input part (recall that $u_i = v_j$), and by assumption on the crossing sequence he received the same message $(c_j(v))_1 = (c_i(u))_1$ (or the same multiple messages if the machine previously returned on the ith cell). Thus his next message in both cases will be the next element of the common crossing sequence, and the argument continues.

Because at each point, the computation follows either the computation of u or v, also this computation halts. This happens either on Alice's or Bob's part, and because both u and v are accepted, Alice and Bob will conclude that $u_1 \dots u_i v_{j+1} \dots v_{|v|}$ is also accepted. □

From now on, we use the binary alphabet and write $\mathrm{PAL}_{\{0,1\}} = \mathrm{PAL}$.

Corollary 12. *Assume U decides* PAL. *If $x, y \in \{0,1\}^n$, $x \neq y$ and $i, j \in [n+1, 3n]$, then $c_i(x0^{2n}x^R) \neq c_j(y0^{2n}y^R)$.*

Proof. We prove the contra positive. Let $u = x0^{2n}x^R$ and $v = y0^{2n}y^R$ with $|x| = |y| = n$.

For $i, j \in [n+1, 3n]$ we have

$$u_1 \dots u_i v_{j+1} \dots v_{4n} = x0^k y^R$$

for some k. If the crossing sequences at i and j are equal, Lemma 11 implies that U accepts $x0^k y^R$, and hence, by assumption on U, this string is in PAL. The reverse of the string is $y0^k x^R$, hence $x = y$. □

Proof of Theorem 9. The total computation time on input x equals the sum of the lengths of $c_i(x)$. Let U be a machine that decides PAL. We use Corollary 12 to show that there exists an n-bit x for which the length of all crossing sequences $c_i(x0^{2n}x^R)$ with $i \in [n+1, 3n]$ exceeds n/e, (e depends on the machine U). This implies the theorem.

Let x be of length n and let d_x be the shortest crossing sequence in the positions $[n+1, 3n]$:

$$c_{n+1}(x0^{2n}x^R), \dots, c_{3n}(x0^{2n}x^R).$$

By Corollary 12, $d_x \neq d_y$ whenever $x \neq y$. Now consider the set of all shortest crossing sequences corresponding to x of length n:

$$\left\{d_x \colon x \in \{0,1\}^n\right\}$$

This set contains precisely 2^n elements. Therefore, it must contain at least one long sequence d_x, because the number of different crossing sequences of size at most ℓ equals

$$\sum_{i=0}^{\ell} |Q|^i \leq (1 + |Q|)^\ell .$$

Let x be such that $2^n \leq (|Q|+1)^{|d_x|}$. Thus $|d_x| \geq n/e$ for some e that does not depend on n. By definition of d_x, all crossing sequences $c_{n+1}(x0^{2n}x^R)$, …, $c_{3n}(x0^{2n}x^R)$ have length at least $\ell \geq n/e$. Thus, $\sum_{i=1}^{\infty} |c_i(x)| \geq 2n^2/e$ and the computation time is proportional to the squared length of $x0^{2n}x^R \in$ PAL. □

Exercise 7. With essentially the same proof, we can show that if U is a single tape machine that decides the languages

$$\left\{zz \colon z \in \{0,1\}^*\right\}$$

than there is a c and infinitely many strings z such that $U(zz)$ computes at least $|z|^2/c$ steps. Explain where we have to adapt the proof above.

Exercise 8. Let U be a deterministic single tape machine that maps a pair (x, y) to $x + y$, where x and y are interpreted as numbers in binary. Show that for some c and for infinitely many n there are n-bit x, y for which U runs in time n^2/c. *Hint:* use addition to solve the problem of exercise 7 by consider pairs (x, y) where x is the bitwise inverse of y.

Exercise 9. Clearly, PAL is a decidable language. Give 2 proofs that PAL is not a regular language by using (1) the pumping lemma and (2) Theorem 9.

Exercise 10. This exercise is harder, and intended for students who need a challenge. Show that if a language can be decided by a Turing machine that on input x runs in time $o(|x| \log |x|)$, then the language is regular. *Hint:* Show that for long x there must be 3 equal crossing sequences. Then construct 2 shorther strings and repeat the decomposition.

5 Universal Turing machines

It is easy for us to believe that if a function can be programmed in some programming language, then it can be implemented in one of the well known languages used for general programming, such as C++, Java or Python. In this section we hope to convince you that Turing machines have the same power. We show that they can simulate *finite register machines*. Then we argue that typical features of programming languages can be implemented in this language. Finally we show that there are Turing machines that can simulate any other Turing machine given an argument that contains a description of a Turing machine.

In 1936, when Alain Turing made the first mathematical construction of such machines, many people where surprised that this was possible. They did not expect that mechanical computation could be used in such a flexible way. Nowadays, for people who can program or are aware of applications of artificial intelligence, this is not surprising. Some specialists claim that this idea was important in the early history of computer science, because it inspired people to write programs that write and rewrite other programs, like compilers.

What might be surprising nowadays, is that there exist such universal Turing machines that are very simple. Currently, such machines are known with only $s = 6$ states that uses a $t = 4$ symbol tape. (Some simple transformation of the input of the simulated machine is needed for this construction.) This machine was constructed by Y. Roghozin. It is an open question whether there exists such a machine for which $s + t < 10$. The current lower bound for such machines is $s + t \geq 4$.[1]

5.1 Computation with finitely many registers

We study machines that manipulate finitely many *registers*, i.e., variables with values in $\mathbb{Z}_{\geq 0} = \{0, 1, 2, \dots\}$. The operation of the machine is specified by a list of numbered commands. Let a and b be variables. A command can be of the following types:

- $a := 0$
- $a := b$
- $a := a + 1$
- $a := a - 1$ (if $a = 0$ then a remains 0)
- `Stop`
- `Goto line` $\langle$number$N\rangle$
- `If` $a = 0$ `then goto line` $\langle$number $N\rangle$ `else goto line` $\langle$number $M\rangle$.

This list contains several redundant commands, they are just listed for convenience. Perhaps you can already guess how the commands operate. Here is an example of a program that implements $f(a, b) = a + b$.

`Input variables:` a, b

`Output variable:` d

[1] D. Woods and T. Neary, The complexity of small universal Turing machines: a survey, Theoretical Computer Science, 410(4-5) p. 443-450, 2009.

1. $c := a$
2. $d := b$
3. If $c = 0$ `then goto line` 4 `else goto line` 5
4. `Stop`
5. $c := c - 1$
6. $d := d + 1$
7. `Goto line` 3

The commands in a program are executed subsequently until a `Goto` or `If` statement is reached. We explain the latter. If the variable in the if statement is zero, then the computation is resumed at line number N, otherwise, at line number M. The input of a register machine is stored in one or more variables. Initially, all other variables are zero. After a stop line is reached, the output is obtained from a specified variable.

Note that addition of two n-bit binary numbers requires $O(n)$ time on a Turing machine. Here, it requires $O(2^n)$ time, hence the construction is highly inefficient.

Exercise 11. Show that the set of functions that can be evaluated by these machines does not change if we remove both the copy command $a := b$ and the reset command $a := 0$.

In the example above the program does not change the value of its input variables. If a function f can be implemented in this way, we can extended the set of instructions with lines $u := f(a, b, \ldots, e)$. The extended language implements the same functions, because each such a line can be expanded to the code of the function, provided we adapt the line numbers and adapt the names of the variables that are used twice. Now it is clear how to write programs for subtraction, multiplication, devision, remainder (mod operation), exponentiation, prime testing, computing the n-th prime number, etc.

In the next chapter we use the following exercise.

Exercise 12. Show that a program for any register machine can be rewritten as a program that only uses the following three types of commands:

- $a := a + 1$
- `If` $a = 0$ `then goto line` m `else` $a := a - 1$ `and goto line` n
- `Stop`

5.2 Turing machines can simulate register machines

We show that register machines and Turing machines can evaluate the same functions. For this, we need to associate bitstrings with the numbers $0, 1, 2, \ldots$ For this we use the sequence from the first example of section 2

$$\begin{array}{rcl} \varepsilon & \leftrightarrow & 0 \\ 0 & \leftrightarrow & 1 \\ 1 & \leftrightarrow & 2 \\ 00 & \leftrightarrow & 3 \\ 01 & \leftrightarrow & 4 \\ & \ldots & \end{array}$$

Theorem 13. *For every register machine there exists a Turing machine that computes the same function.*

Proof. We give the proof for one argument functions. The idea is to use the Turing machines from the examples in section 2 to implement the increment $a := a+1$ and decrement $a := a-1$ commands. By applying the construction of Lemma 6 we obtain machines that never leave the tape.

We assume that the register machine uses the same variable for input and output; if this is not the case, we copy the output variable to the input variable. In exercise 11 it is shown that the copy and reset commands are redundant, so we can modify our program such that these lines do not appear. If the program of the register machine uses k variables, then we use a Turing machine with k-tapes. Each tape stores one variable and the 1st tape is used for the input-output variable.

For each line ℓ of the program, the computation head contains some sufficient number of states $q_{\ell,1}, \dots, q_{\ell,e}$ to execute the following instructions.

- If line ℓ contains a command `Goto line` n, then the list contains a single state $q_{\ell,1}$ and regardless of the content of the tapes, the subsequent state is $q_{n,1}$.
- If a line ℓ contains a command $a := a + 1$, then the machine of section 2 is run on the tape that contains a. If this machine halts, then the computation head switches to the state $q_{\ell+1,1}$. Similar for a decrement operator.
- If line ℓ contains an `If` command, $q_{\ell,1}$ checks whether the first cell of the variable's tape is blank and transfers to $q_{N,1}$ or $q_{M,1}$ accordingly. Here, N and M are the line numbers indicated in the `then` and `else` parts.
- If the ℓth line is a `stop`-line, then $d(q_{\ell,1}, a_1 \dots a_k)$ is undefined for all $a_1 \dots a_k \in \Gamma^k$.

It is easy to understand that this machine simulates the register machine. □

5.3 Register machines can simulate Turing machines

In general, it is easier to implement functions in this language compared to programming Turing machines, and therefore, it is easier to believe that any algorithm can be programmed in it. However, the language lacks arrays. It turns out that this can be implemented too: an array $[a, b, c, d, e]$ can be stored as $A = 2^a 3^b 5^c 7^d 11^e$. The commands $A[i] := k$ and $f := A[i]$ can now be replaced by small programs with input variables A, i, k and f, A, i. (In particular these programs will include the computation of the nth prime number for a given n.) In fact, we can code an infinite sequence of nonnegative integers in this way, provided this sequence is initialized with all zeros, because a zero at position e contributes a factor $(p_e)^0 = 1$.

Exercise 13. Show that the class of functions which can be implemented in this language remains the same if we restrict the number of variables to a large enough constant, say 100.

Theorem 14. *Every partial computable function can be evaluated by a machine with finitely many registers.*

Proof. We assume that we have arrays available as explained above. Let (Q, Γ, d, q_0) be a deterministic Turing machine with one tape. We can code the tape in a list variable T and access the ith cell using $T[i]$. We associate every symbol in Γ with a number $0, 1, \dots, |\Gamma| - 1$; we choose 0 for the $\sqcup$ symbol. Because at each computation step, there is only a finite part of the tape that is used, at most finitely many elements in the array are nonzero, and at each point, the full tape is represented by a finite product.

We first explain how to simulate a computation step. For this, we maintain four variables:

- q encodes the state of the computation head,
- i encodes the position of the head,
- T encodes the tape (as an array),
- b encodes the current tape symbol that is scanned.

The "main loop" of the program updates these values. There are many ways to implement the main loop, just verify for yourself there exists at least one way.

Before entering the main loop, we must transform the input. Initially, each input string is represented as its index in the list of all strings in the order above. Assume that we associate $0 \in \Gamma$ with $1 \in \mathbb{N}$ and $1 \in \Gamma$ with $2 \in \mathbb{N}$. It is a bit tedious but not to hard to find a program that transforms $x = x_1 \dots x_e$ the number $2^{1+x_1} 3^{1+x_2} \dots p_e^{1+x_e}$. Then we run the simulation above. After a halting state is reached, we must capture and transform the output. Again, verify for yourself there is at least one way to program this. □

Theorem 15 (Universal Turing machine.)**.** *There exists a partial computable function $\varphi\colon \{0,1\}^* \times \{0,1\}^* \to \{0,1\}^*$ such that for each partial computable function $g\colon \{0,1\}^* \to \{0,1\}^*$ there exists a string w such that g equals the partial function $x \mapsto \varphi(w, x)$.*

Proof idea. The previous two theorems state that register machines and Turing machines compute the same functions. Therefore, it suffices to show that a register machine can be encoded in an integer, and that there exists a register machine that can simulate any other register machine given the corresponding integer as additional input. There are many ways to do this. For example, we may implement 2 dimensional arrays and store the description of such a machine in such an array: all information in a command line can just be described by a few numbers. We store all variables in a one-dimensional array. Then construct a main loop to update the line number and the variables. □

6 Some decidable languages

From now on, we present algorithms as usual, using high level descriptions.

Exercise 14. Assume that B and C are decidable. Show that also $B \cup C$, $B \cap C$, $B \setminus C$, BC and B^* are decidable. Is every finite set decidable?

We define decidable sets for pairs of strings using some standard pairing function. Natural numbers $\mathbb{N} = \{1, 2, \dots\}$, integers, and tuples of such objects can all be represented using bitstrings, and we assume that we have fixed such a representation. Rational numbers can be interpreted as a pair in $\mathbb{N} \times \mathbb{Z}$, and can therefore also be represented.

Exercise 15. Consider the set of all n for which the decimal representation of $\pi = 3.14159\ldots$ contains at least n nines in a row.

$$\begin{aligned}\pi = {} & 3.1415\mathbf{9}265358\mathbf{979}32384626433832\mathbf{7}\mathbf{9}50288841\mathbf{9}716\mathbf{9}3\mathbf{99}375105820\mathbf{9}74\mathbf{9}445\mathbf{9}2307816406 \\ & 28620 8\mathbf{99}862803 \quad \ldots \quad 747713 0\mathbf{99}605187072113 4\mathbf{999999}8372\mathbf{9}7804\mathbf{99}5105\mathbf{9}73173 \quad \ldots\end{aligned}$$

For example, the 2nd part of π's decimals representation above[2] contains the substring 999999, hence S contains $1, 2, \ldots, 6$. Show that this set is decidable. (Note that the proof is nonconstructive.)

Exercise 16. Show that an infinite set of natural numbers is decidable if and only if it is the image of an increasing computable function, i.e., if there exists a computable increasing $f\colon \mathbb{N} \to \mathbb{N}$ such that $S = \{f(1), f(2), f(3), \ldots\}$.

Exercise 17. Show that the set of rational numbers smaller than $\sqrt{2}$ is decidable. Do the same for the irrational number $e = \sum_{i=0}^{\infty} \frac{1}{i!}$ (the base of the natural logarithm).

Exercise 18. Show that every partial computable $f\colon \mathbb{N} \to \mathbb{N}$ has a partial computable *pseudo inverse*, i.e., a function g for which $f(g(f(x))) = f(x)$ for all x.

We fix a method to encode an automaton B as a bitstring $\langle B \rangle$. We assume that given such a description $\langle B \rangle$ and a string w, we can simulate the automaton on input w.[3]

We use a generic method to combine descriptions of several objects, and for this we also use the bracket notation. For example, for an automaton B and a string w, we use $\langle B, w \rangle$ for a representation of the pair (B, w). Given such a representation, a Turing machine can start the simulation of B on input w. This implies for example that the set

$$A_{\text{DFA}} = \{\langle B, w \rangle \colon B \text{ is an DFA that accepts } w\}$$

is decidable.

Lemma 16. *The following set is decidable:*

$$E_{NFA} = \{\langle B \rangle \colon B \textit{ is an NFA that accepts no strings}\}$$

Proof. The algorithm that decides the set proceeds as follows. First we use $\langle B \rangle$ to construct the directed graph of all states. Then we search for a vertex in the set of accepting states that can be reached from the start state. We can do this to by checking all paths of length at most $|Q| - 1$, where Q is the set of states. Why is it enough to check paths of length $|Q| - 1$? $\square$

The method described in the proof requires exponential time. Do you know faster methods?

Exercise 19. Show that the following sets is decidable

$$EQ_{\text{NFA}} = \{\langle B, C \rangle \colon B \text{ and } C \text{ are NFAs that accepts the same strings}\}.$$

Exercise 20. Consider the set of all descriptions $\langle M \rangle$ of nondeterministic automata M that accept at least 1 string w that contains a substring 111. Show that this set is decidable.

[2] Obtained from `http:/www.piday.org/million/`

[3] A *reasonable* description of automata, Turing machines, or other computing devices also means that the set of valid descriptions is decidable.

A *linear bounded automaton* is a Turing machine that on input w never moves its computation head further then the $(|w|+1)$th cell of the tape. Such machines are still quite powerful, for example, every regular language can be decided by such an automaton. The following exercise is inspired by such machines.

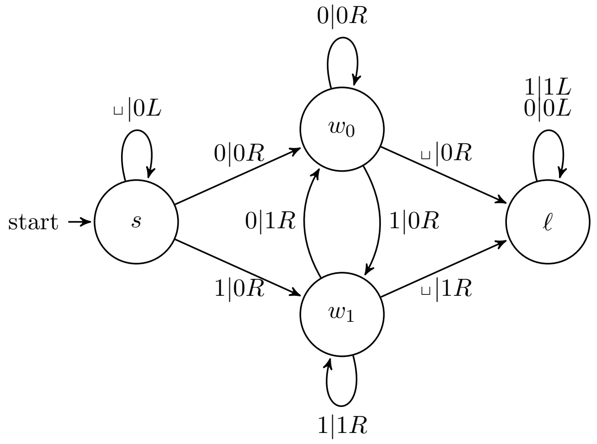
Exercise 21. The set

$$\{\langle M, w\rangle \colon M \text{ is a TM that accepts } w \text{ using at most } |w|+1 \text{ tape cells}\}$$

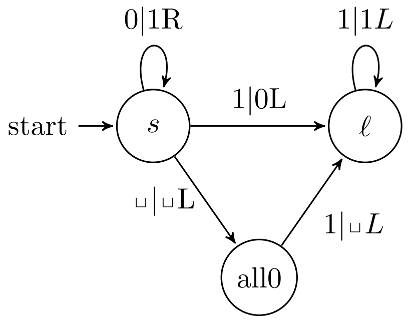
is decidable. Hint: the problem is that one does not know when the machine will halt. Note that the tape and hence the machine can have at most finitely many states.

Solutions

Exercise 1. The machine has 4 states. In the start state we always write 0. In the states w_0 and w_1 the machine always writes 0 and 1. In the state ℓ the machine always moves left until it falls of the tape.



Exercise 2.



Exercise 4. A deterministic automaton scans its input symbol by symbol, and a Turing machine can imitate this behavior by always moving right and using the same states as the automaton to simulates its computation. When it reaches a blank, the Turing machine halts. If we use the same set of accept states, the machine accepts the same strings.

Formally, let (Q, Σ, D, s, F) be a DFA. Recall that $D \subseteq Q \times \Sigma \times Q$ and that for all $(q, a) \in Q \times \Sigma$, there is an outgoing edge. For all $q \in Q$ and $a \in \Sigma$ let

$$d(q, a) = (R, \sqcup, r) \quad \text{for the unique } r \text{ such that } (q, a, r) \in D.$$

It is not important what the machine writes. $d(q, \sqcup)$ is undefined for all $q \in Q$. The Turing machine $(Q, \Sigma \cup \{\sqcup\}, d, s)$ will halt in a state from F precisely for those strings that are accepted by the DFA. Hence, it decides the same language.

Exercise 5. From the solution of the previous exercise, we see that regular languages can be decided by such machines. Note that the tape is useless for these machines, thus the machines in some sense compute without memory. Therefore, we expect that these languages are all regular.

We prove this formally by constructing an automaton for each such Turing machine. The computation has 2 phases, in the first phase, it is reading the input string. In the second phase, it is reading only blanks. At any point, the write operations do not influence the remainder of the computation. Given such a Turing machine (Q, Γ, d, s) and accept states F, we define an NFA of the form $(Q, \Sigma, D, s, \tilde{F})$, where

$$D = \big\{(q, a, r)\colon q \in Q,\ a \in \Sigma \text{ and } d(q, a) = (b, R, r)\big\}$$

To determine whether a state q should be an accept state, we run the Turing machine on a blank tape with q as start state. If the machine reaches a state of F, we put q into $\tilde{F}$. Now we obtained an automaton that recognizes the same language.

Exercise 8. If $|x| = |y| = n$ and x is the bitwise inverse of y, then $x + y = 1^n$. Suppose that for all n and for all inputs uv with $|u| = |v| = n$, we can compute the $u + v$ in time $t(n)$. We use the machine to solve the problem of 7 as follows:

On input a string zw with $|z| = |w|$, we negate the bits of w and compute the sum. Then we check whether the sum equals 1^n, and if this is true we accept. Otherwise, we reject.

Note that this works, the sum equals 1^n if and only if $z = w$. The total computation time of the algorithm is at most $t(n) + en$ for some e and all large n. By exercise 7 for every machine, there is a c and infinitely many $z \in \{0,1\}^*$ for which $t(n) + en \geq n^2/c$ with $n = |z|$.

Exercise 9. (1) Note that $0^n 1^n 0^n \in$ PAL. Assume the language were regular. Consider this string for n being the pumping length. After pumping up or down, we obtain a string of the form $0^{n+j} 1^n 0^n$ with $j \neq 0$. This is not a string in PAL, contrary to what we conclude from the pumping lemma. Hence, our assumption must be false, and PAL is not regular.

(2) All regular languages can be decided using n computation steps on inputs $x \in \Sigma^n$. For this, we use a Turing machine that runs once over the input and simulates some DFA that recognizes the language. But this contradicts Theorem 9.

Exercise 15. By definition, if the set contains some n, then it contains also $1, 2, \cdots, n-1$. Thus, there are 2 cases: either for all n, the substring

$$\overbrace{99\ldots9}^{n \text{ times}}$$

appears in the decimal expansion, or not. In the first case, we have the set equals $\mathbb{N}$ and this set is clearly decidable. Otherwise, the set is finite hence decidable. In both cases the set is decidable. But we do not know which case it is true, and hence, we can specify a program that decides the set.

Exercise 16. Suppose that a set S is infinite and decidable. Consider the algorithm that on input n, searches for n different elements in S by running the decision procedure to the natural numbers in increasing order. This search is always terminates, because S is infinite. After this, we output the largest element. By construction, this function is total, increasing and computable.

Suppose f is an increasing computable function and let $S = \{f(n) \colon n \in \mathbb{N}\}$. On input n, the decision procedure for S searches for an argument m such that $f(m) \geq n$. It accepts if $n \in \{f(1), \ldots, f(m)\}$, and otherwise it rejects. Note that the search for m is always successful. Hence, this algorithm indeed decides S.

Exercise 17. For the first part, simply compute the square and compare it with 2. To decide whether $r < e$, we search for a value $k \geq 1$ such that $r - \sum_{i=0}^{k} \frac{1}{i!}$ is either negative or exceeds $\frac{1}{k!}$. In the first case we accept in the second case we reject.

We show that this works. Clearly if the algorithm accepts, we have $r < e$. Note that for $k \geq 2$, we have that

$$\frac{1}{k!} \geq \sum_{j=1}^{\infty} \frac{1}{(k+j)!} \ ,$$

because each term in the sum is at most half of the previous term. Hence, if the algorithm rejects, we have $r > e$. If the algorithm never accepts or rejects, this means that $r = e$, which is impossible because e is irrational.

Exercise 19. Let L_B and L_C be the languages recognized by B and C. Note that the symmetric difference $(B \setminus C) \cup (C \setminus B)$ is also regular. Moreover, we can compute an automaton that recognizes this languages: first we construct equivalent deterministic automata using the algorithm of the proofs that DFA and NFA recognize the same languages. Now we can apply the construction for complements. Then we apply the constructions for intersections and unions. After the automaton is obtained, we apply the algorithm obtained from Lemma 16 to check whether this set is empty. If this is the case, we accept $\langle B, C \rangle$, otherwise we reject.

Exercise 21. We need to simulate M on input w and check whether M outputs 1. The problem is that M might get stuck in a loop, and we do not know when to halt the simulation.

However, if $M = (Q, \Gamma, d, s)$ never moves behind the $(|w| + 1)$th cell, then either the computation halts, or some machine state will be visited twice, and both events can be detected.

Indeed, the tape can have at most $|\Gamma|^{|w|+1}$ states. The number of machine states in $\mathbb{N} \times Q \times \Gamma^{\infty}$ in a computation path is is at most $N = (|w| + 1)|Q| \cdot |\Gamma|^{|w|+1}$. Thus, if a computation path is unbounded, some state will appear twice.

In fact, it suffices to run the computation during N steps. If the machine did not halt, then it will never halt, because some state must have been visited twice, and hence the computation goes in a loop.