Turing machines and computable functions[*]

In this chapter we address the question: "how can we define the class of discrete functions that are *computable*?" 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: it's full specification is too long, and someone else might insist on using his own favourite language (imagine I am an old professor who loves the COBOL language).

Turing machines present a solution. The Church-Turing thesis states that every discrete mechanical process can be simulated by a Turing machine. This claim has been justified by showing that Turing machines can simulate the operation of any interesting and reasonable model for discrete computation.

In this chapter we define Turing machine. We show that such machines can implement typical features of programming languages. We argue that the definition is robust: changing the tapes (having more tapes, increasing the alphabet), or switching to a nondeterministic computation, does not change the class of computable functions. We show that there are problems that can be solved much faster if the machine has an additional work tape. Finally we show that there exists a universal Turing machine, i.e., a machine that can simulate any other machine.

The notes below are long and detailed. 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, (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

A Turing machine is a computing device that on input a finite sequence of strings might produce a string. The machine has two parts: a control unit and a tape. The control unit is characterized by a finite set Q of *control states*, a *start state* $q_0 \in Q$ and a *transition function* δ. The tape is a one-way infinite sequence of cells and each cell contains a symbol from a 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 the start state q_0. 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,

[*]Theoretical Computer Science 2016-2017, National Research University Higher School of Economics. Version: November 20th, 2016. Thanks to Kirill Neklyudov for reporting some mistakes.

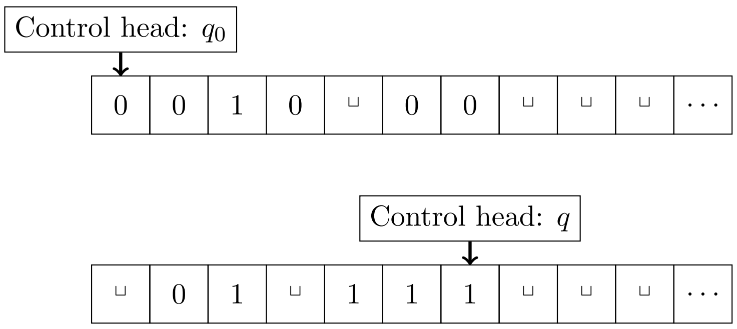


Figure 1: The initial configuration for the computation $U(0010, 00)$ and a random configuration.

depending on the current state and the symbol below the control head, the transition function, 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 tries to move 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.

More precisely, the transition function is a function $\delta : Q \times \Gamma \to \Gamma \times \{-1, 1\} \times Q$. A value $\delta(q, a) = (b, m, r)$ 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 $a^\infty = aa\dots$. Let $\sqcup$ be a fixed symbol. $\mathbb{N} = \{1, 2, 3, \dots\}$.

At each computation step, the state of the Turing machine is fully described by the position $i \in \mathbb{N}$ of the computation head, the state $q \in Q$ of the computation head and the content $T \in \Gamma^\infty$ of the tape, i.e., by $(i, q, T) \in \mathbb{N} \times Q \times \Gamma^\infty$.

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

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

if and only if $\delta(q, T_i) = (b, m, r)$. There are no other edges.

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

We also use Turing machines that recognize a language. In the part of computational complexity we use the non-deterministic variant of such machines.

Definition 2. *A* non-deterministic Turing machine *is a 4-tuple (Q, Γ, D, q_0) where $q_0 \in Q$, $\sqcup \in \Gamma$, Q, Γ are finite sets, and $D \subseteq Q \times \Gamma \times \Gamma \times \{-1, 1\} \times Q$. A Turing machine defines a graph where the vertices $\mathbb{N} \times Q \times \Gamma^\infty$ are called* states*. There is a directed* edge *from (i, q, T) to (j, r, S) if $T_k = S_k$ for all $k \neq i$ and*

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

A recognizer *is a 5-tuple (Q, Γ, D, q_0, F) where (Q, Γ, D, q_0) is a non-deterministic Turing machine and $F \subseteq Q$. A recognizer* accepts *$w \in \Gamma^*$ if there exists a path from $(1, q_0, w\sqcup^\infty)$ to an element of $\mathbb{N} \times F \times \Gamma^\infty$. A recognizer is a* decider *over Σ if for all $w \in \Sigma^*$ there is no infinite path leaving from $(1, q_0, w\sqcup^\infty)$.*

2 Examples

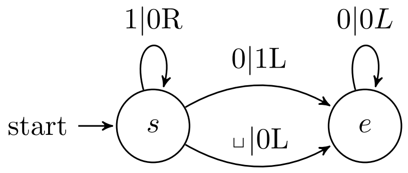
We give some detailed examples of Definition 1. These examples will be used in the construction of a universal Turing machine.

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 precisely 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 $(\{s, e\}, \{0, 1, \sqcup\}, \delta, s)$ where δ is defined by

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



In the table 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	$\sqcup$	$\ldots$

The state $(1, e, 0000\sqcup^\infty)$ has no subsequent state, because $(0, e, \cdot) \notin \mathbb{N} \times Q \times \Gamma^\infty$. Hence, the computation halts and the output is 0000 as desired.

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

Exercise 2. Show that the class of computable functions remains the same if we consider machines that never try to move from their tape, it is: if the computation head is in the leftmost cell, it either halts or moves right.

Expanding the input. Let $a \in \Sigma$ be a symbol. The function $f : \Sigma^* \to \Sigma^* = x_1 a x_2 a \ldots x_{|x|} a$ is computable. Probably, you can already implement a TM yourself to prove this claim. Here is my solution.

The idea to implement the machine is to first insert the leftmost a symbol, and thus move $x_2 \ldots x_{|x|}$ one position to the right. After this move, the machine needs to return to x_2 and insert an a to the right of x_2 and hence move $x_3 \ldots x_{|x|}$ and so on. This approach has a problem: when the machine needs to return, it does not know where it was last because the input can contain a letters. For this it inserts a symbol $\dot{a} \notin \Sigma$, rather than a. After moving the remainder of the input one cell to the right, the machine returns to the $\dot{a}$ symbol, replaces it by a, skips a letter and then again inserts $\dot{a}$. Assume $\sqcup \notin \Sigma$. The machine is given by

$$\left(\{s, l, j, j'\} \cup \{r_a : a \in \Sigma\}, \Sigma, \{\sqcup, *, \dot{a}\} \cup \Sigma, \delta, s\right),$$

where δ is given by the table in figure 2. In this table b, b' can take all values in Σ. Note that on input ε the machine will start above a blank and directly terminate.

With a similar technique, we can insert a string aa or a^k for some $k \geq 1$ after each letter of the input string. One can also construct a machine that does the reverse: compressing the input by deleting all bits at positions different from 1 modulo k.

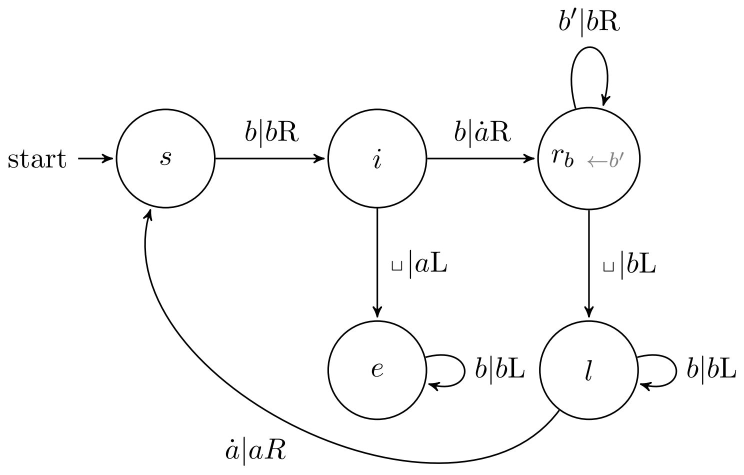
3 A robust definition of computable functions

Definition 3. *A partial function $f : \Sigma^* \to \Sigma^*$ is* partial computable *if and only 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 $f(x) = U(x)$. Similar for functions $f : \Sigma^* \times \Sigma^* \to \Sigma^*$, $f : (\Sigma^*)^3 \to \Sigma^*$, etc. A function is* computable *if it is total and partial computable.*

A Turing machine decides *a language L over Σ, if it is a decider over Σ and it accepts precisely the strings of L.*

These definitions do not change if we use Turing machines with two-way infinite tapes, multiple tapes (see further), and even two-dimensional tapes, or restrict the tape alphabet to $\Gamma = \Sigma \cup \{\sqcup\}$. We can also consider machines that in each computation step either writes or moves. However, some changes can be dangerous: if the machine can not move left, the tape will become practically useless.

The characteristic function of a set L is the binary function I_L for which $I_L(x) = 1$ if $x \in L$ and $I_L(x) = 0$ otherwise. Note that $L \subseteq A^*$ is decidable if and only if its characteristic function is computable.



q, a	$\delta(q, a)$	
s, b	$\dot{a}, R, i$	skip a letter
i, b	$\dot{a}, R, i$	insert $\dot{a}$
$i, \sqcup$	a, L, e	terminate after seeing $\dot{a}\sqcup$
r_b, b'	$b, R, r_{b'}$	move input to the right
$r_b, \sqcup$	b, L, l	return to the last inserted $\dot{a}$
l, b	b, L, l	
$l, \dot{a}$	a, R, s	replace $\dot{a}$ by a and repeat.
e, b	b, L, e	return to the beginning of the tape.

Figure 2: Expanding the input

Exercise 3.

1. Show that 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.

2. Show that if f and g are computable then also the function $t \to f(g(t))$ is computable. Use exercise 2.

3. Show that regular languages are decidable.

4. What is the set of languages recognized by machines that can not move their computation head to the left?

Theorem 4. *The class of computable functions in $\{0,1\}^*$ does not change if we only use Turing machines whose tape alphabet is $\{0,1,\sqcup\}$.*

Proof. In the previous section we described Turing machines at the formal low level. In this proof and further, we will describe machines at the informal low level, i.e., we describe the movements of the computation head, but do not bother to write the table of the transition function.

We construct a machine $U' = (Q', \{0,1,\sqcup\}, \delta', q_0')$ that simulates a machine (Q, Γ, δ, q_0). The idea is to virtually partition the tape in k-bit blocs for some $k \geq \log |\Gamma|$, and encode each letter by a k-bit string. Choose for the $\sqcup$, 0 and 1 symbols the words $\sqcup^k, 0\sqcup^{k-1}$ and $1\sqcup^{k-1}$.

In the beginning of the computation with input w, machine U' transforms its tape from $w\sqcup^\infty$ to $w_1\sqcup^{k-1}\ldots\sqcup^{k-1}w_{|w|}\sqcup^\infty$. For this, it runs the machine from the second example of the previous paragraph. A computation $U'(w)$ must therefore start with transforming w in this form. If there are more arguments, the transformation is similar.

In each computation step U first reads, then writes, then moves, and finally changes its state. Hence, U' reads k letters, then returns $k-1$ cells to the left, writes k letters, then moves either moves $2k-1$ cells to the left or one cell to the right. During the k reading steps, the computation unit must remember what was red. Let $\{0,1\}^{\leq k}$ be the set of bitstrings of length at most k. For the reading operation, we add a copy of $Q \times \{0,1\}^{\leq k}$ to Q'. These states remember the part of the block of k cells that was red. Hence, $\delta'((q,x),a) = (a, R, (q, xa))$. Then to return $k-1$ positions, we add a copy of $Q \times \{0,1\}^{\leq k} \times \{1,2,\ldots,k-1\}$ to Q', because the bits that were red need to be remembered. And so on.

When the computation of U terminates, we need to transform the output back to its compressed form. This happens in a similar way as the decompression of the input. □

Exercise 4. (Linear speedup theorem.) Let $f : \Sigma^* \to \Sigma^*$ be a total function that can be evaluated by a Turing machine in time $t : \mathbb{N} \to \mathbb{N}$ and let e be a constant. Show that there exists a Turing machine that computes f in time t' with $t'(n) = t(n)/e + O(n^2)$.

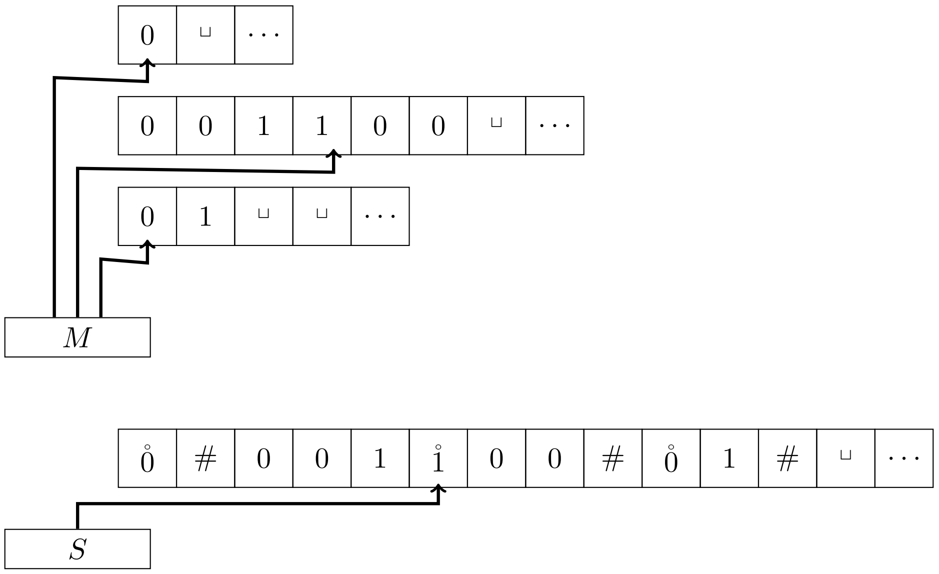
4 Multitape Turing machines

A k-tape Turing machine is an ordinary 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 appears 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 and all move in an individual direction. Afterwards, the control unit transits to its new state.

The control unit of a k-tape TM is described by a partial function

$$\delta : Q \times \Gamma^k \to \Gamma^k \times \{-1, 1\}^k \times Q.$$

The computation terminates if one of the read-write heads tries to go of the tape, or if δ is undefined in the state. The output is defined in a similar way on the first tape. We do not repeat the formal definition because it is so similar; we just mention that states are given by $\mathbb{N}^k \times Q \times (\Gamma^k)^\infty$.



0 ⊔ ···
0 0 1 1 0 0 ⊔ ···
0 1 ⊔ ⊔ ···
M
$\mathring{0}$ # 0 0 1 $\mathring{1}$ 0 0 # $\mathring{0}$ 1 # ⊔ ···
S

Figure 3: Definition of a multitape Turing Machine.

4.1 Mutlitape Turing machines compute the same functions

Theorem 5. *A partial function can be evaluated on a multitape Turing machine if and only if it can be evaluated on 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 a new # symbol, see figure 3. S's tape alphabet contains two copies of M's alphabet, one copy we denote with dotted symbols. These dotted symbols are used for cells above which the reading/writing head is located.

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

1. First the input is transformed to the form

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

   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 M performs a right move to an unscanned blank cell, a dotted symbol needs to be placed on a cell that contains a # symbol. If this happens, S writes a dotted blank and moves the remaining tape contents one cell to the right.

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

S simulates M. The theorem is proven. $\square$

4.2 Multitape machines are faster than single tape machines

Definition 6. *Let $f : \mathbb{N} \to \mathbb{N}$. A TM U runs in* time f *if for each input x of length n, every computation path has length at most $f(n)$. A TM U runs in* space f *if for each input x of length n, every path contains only elements (i, q, T) with $i \leq f(n)$.*

The proof of Theorem 5 implies a stronger result: one can simulate a multitape machine that runs in time $t(n)$ by a single tape machine that runs in time $O(t(n)^2 + n)$. We show that this quadratic increase is necessary.

Palindromes are words that are the same when spelled backwards. In English one has the following examples: deleveled, kayak, level, madam, racecar, radar, rotator, testset. (If punctuations and spaces are ignored: "Was it a car or a cat I saw", "Dammit, I'm mad!".) Let x^R be the reverse of a string x, i.e., $x^R = x_{|x|} x_{|x|-1} \dots x_1$.

Exercise 5. Show that the language PAL $= \{x : x = x^R\}$ can be decided in time $O(n)$ on a 2-tape Turing machine.

Theorem 7. *For any deterministic single tape machine U that decides PAL, there exists a c and infinitely many x such that the computation time of $U(x)$ exceeds $|x|^2/c$.*

Proof. Fix a machine U. Let the crossing sequence $c_i(x)$ be the list $(a_1, q_1), (a_2, q_2), \dots, (a_e, q_e)$ of elements in $A \times Q$ describing the state of the machine just before the control head arrives at the ith cell of the tape, see figure 4.2.

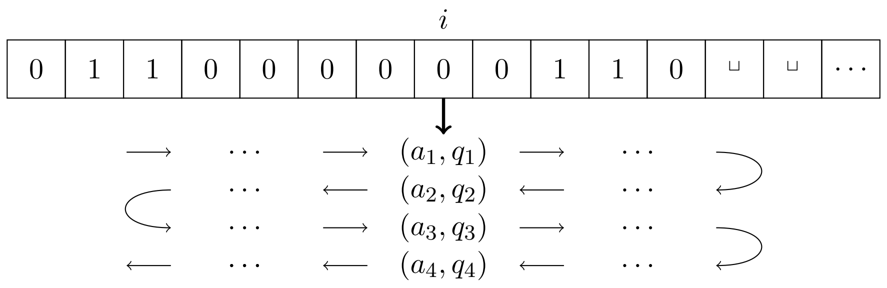


Figure 4: Crossing sequence $c_i(x)$

Lemma 8. *If $c_i(u) = c_j(v)$, and U accepts both u and v, then U accepts*

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

Proof. Indeed, as long as the computation head is at the left of position i, it follows the accepting computation of u. Each time the computation head is at a position which is at least i, it follows an accepting computation for v. Hence, it must terminate on one of the sides, and because it accepts both u and v, it will also accept the string above. □

Corollary 9. *Let U be a machine that decides PAL. If x and y are different n-bit strings and $i, j \in [n+1, 3n]$, then $c_i(x0^{2n}x^R) \neq c_j(y0^{2n}y^R)$.*

Proof. Indeed, $xy^R \notin$ PAL, hence $x0^k y^R \notin$ PAL for all k, because after deleting a character in the middle of some string in PAL, the string remains in PAL. Let $u = x0^{2n}x^R$ and $v = y0^{2n}y^R$. 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 were equal, then by Lemma 8 the automaton would also accept this string, which is outside PAL. □

We finish the proof of Theorem 7. 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 9 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 only on the machine U). This implies the theorem.

Fix some large n and for each n-bit x let d_x be the shortest crossing sequence among

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

By Corollary 9, $d_x \neq d_y$ whenever $x \neq y$. Let ℓ be the maximal length of a crossing sequence d_x of an n-bit x, i.e., $\ell = \max\{d_x : |x| = n\}$. Let us count the number of different crossing sequences of size less then ℓ:

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

Because all crossingsequences d_x are different, we conclude that $2^n \leq (|Q|\,|\Gamma|)^\ell$, i.e., $n \leq e\ell$ for some e that only depends on U. Hence, there is an x for which all crossingsequences $c_{n+1}(x0^{2n}x^R), \dots, c_{3n}(x0^{2n}x^R)$ have length at least $\ell \geq n/e$. □

Exercise 6. How much do you need to adapt the proof of Theorem 7 so that it works for non-deterministic Turing machines?

Exercise 7. Let U be a deterministic single tape machine that decides the set of all triples $(x, y, z) \in \{0,1\}^* \times \{0,1\}^* \times \{0,1\}^*$ such that $x+y = z$ when interpreted as numbers in binary. Show that for some c and for all n there are n-bit x, y, z for which U runs in time n^2/c.

5 Universal Turing machines

In the next lesson we show that the Halting set is not decidable. For this and many other results, we need Turing machines that can simulate any other Turing machine. That this is possible, is no surprise to us from our experience in real life with virtual machines (and artificial intelligence). Universal Turing machines are a mathematical model for such machines. In 1936, when Alain Turing made the first mathematical construction of such machines, many people where surprised by the idea that a single machine can do what all other machines do. Several scientists today claim that the result and its proof was important in the early history of computer science, because it inspired people to write programs that write programs, like compilers do.

What might be surprising, is that there exists a Turing machine with only $s = 6$ states that uses a $t = 4$ symbol tape that can simulate any other machine after a simple encoding of the input. 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] We construct a universal machine using machines that have finitely many registers. These machines are easy to program and will be used to prove Gödel's theorem.

5.1 Computation with finitaly many registers

We study machines that manipulate finitely many variables, called *registers*. These registers take values in $0, 1, 2, \ldots$ The operation of the machine is specified by a list of numbered commands. Let a and b be variables. A command can have the following types:

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

This list contains several redundant commands, they are just listed for convenience. 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. 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 survay, 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

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 8. Show that the set of functions that can be evaluated by these machines does not change if we remove the copy command.

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 original 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, test for primality, computing the n-th prime number, etc.

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

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

Theorem 10. *For every register machine that computes a function $f : \mathbb{N} \to \mathbb{N}$, there exists a Turing machine that computes the same function. The same holds for functions f with two or more arguments.*

Proof. The idea is to use the Turing machines from the first example in 2 to implement to implement commands $a := a + 1$ and from 2 to implement $a := a - 1$. We assume that the machines never try to go off their tape, for this, we use the construction of exercise 2.

Consider a register machine that implements an $f : \mathbb{N} \to \mathbb{N}$ such that the input and output variable are the same. If the program of the register machine uses m variables, then we use a Turing machine with m-tapes. Each tapes stores one variable. We also assume that the computation heads are allowed to stay in the same position during a computation step.

For each line ℓ of the program the computation has a list of states $q_{\ell,1}, \ldots, q_{\ell,e} \in Q$.

- 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_ℓ checks whether the initial cell of the tape of the variable is empty and transfers to the state of the corresponding line.
- If a line contains the command $a := 0$, the tape is cleaned.
- If the ℓth line is a `stop`-line, then δ is undefined in $q_{\ell,1}$, regardless of the tape content.

In exercise 5.1 it is shown that the copy command is redundant. It is easy to understand that this machine simulates the register machine. It is easy to adapt the proof for functions with more arguments. The theorem is proven. □

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 $2^a 3^b 5^c 7^d 11^e$. The commands $a[i] := b$ and $b := a[i]$ can now be replaced by small programs with input variables a, b, i. (In particular these programs will include the computation of the nth prime number for a given n.)

Exercise 9. 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 11. *Every partial computable function can be evaluated by a machine with finitely many registers.*

Hence, if we want to show that an other computation device can simulate a Turing machine, it is sufficient to show that it can simulate any register machine.

Proof. We assume that we have arrays available as explained above. Let (Q, Γ, δ, 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 $0, 1, \ldots, |\Gamma| - 1$; 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 hence, the full tape can be encoded using a product of prime numbers that remains finite.

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 this to a number $2^{1+x_1}3^{1+x_2}\dots p^{1+x_e}$. 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. $\square$

5.2 Construction

Below we will use subscript notation for the first argument of a two argument function: $\varphi_w(x) = \varphi(w, x)$. If one argument is fixed we simply write φ_w for the function $\varphi(w, \cdot)$. For all $k \geq 2$, we can do the same with k-argument functions.

Theorem 12 (Universal Turing machine.)**.** *There exists a partial computable function $\varphi : \{0,1\}^* \times \{0,1\}^* \to \{0,1\}^* : (w,x) \to \varphi_w(x)$ such that for each partial computable function $g : \{0,1\}^* \to \{0,1\}^*$ there exists a w such that $g = \varphi_w$.*

Exercise 10. Prove Theorem 12. Because register machines and Turing machines compute the same functions, it suffices to show that all register machines with one input and output variable can be encoded as a natural number n, and that there exists a register machine that on input n and m simulates the machine n with input m.

In the part about Kolmogorov complexity, we need a Turing machine that is not only universal, but simulates any other machine by prefixing some string on its first argument.

Definition 13. *A k-argument partial computable function φ is called a* prefix Gödel numbering *if for every k-argument function ψ there exists a string s such that $\psi_w = \varphi_{sw}$ for all w and x.*

Note that by Theorem 12, a prefix Gödel numbering contains an index for every partial computable function.

Theorem 14. *There exists a prefix Gödel numbering.*

Proof. We prove the case $k = 2$, the other cases are similar. For the same reasons as in the proof of Theorem 12 there exists a $(k+1)$-argument function ϕ such that for each partial computable k-argument function ψ there exists a w such that $\phi_w(x,y) = \psi(x,y)$. We code the first two arguments of ϕ in a single argument using a prefix-free code. For example, we can code $w = w_1 \dots w_n$ as $\hat{w} = 0w_1 0w_2 \dots 0w_n 1$. Let

$$\varphi_u(y) = \begin{cases} \phi_w(x,y) & \text{if there exist unique } w \text{ and } x \text{ such that } u = \hat{w}x, \\ \text{undefined} & \text{otherwise.} \end{cases}$$

If it exists, the decomposition of u is unique and $\varphi_{\hat{w}x}(y) = \phi_w(x,y)$. This is because the set of all $\hat{z}$ for all z is prefix-free. Moreover, a register machine for ϕ can easily be transformed to a machine for φ. $\square$

5.3 Transforming programs

The results above are useful to reason about programs that manipulate programs, or Turing machines that manipulate descriptions of Turing machines. We consider programs for a prefix Gödel numbering φ. For example, imagine the function that maps a program p evaluating $f : \{0,1\}^* \to \{0,1\}^*$ to a program p' for the function $x \to f(f(x))$. The theorem above says that the transformation is computable, and in fact very simple: there exists a string s such that we can choose $p' = sp$. Why? The function

$$\psi_p(x) = \varphi_p(\varphi_p(x))$$

is partial computable, hence there exists an s such that $\varphi_{sp}(x) = \psi_p(x) = f(f(x))$ for all p and x.

Another example is the transformation that hardwires a fixed value w for an argument in a program. In otherwords, for each p, let p_w be the program that on empty input evaluates $\varphi_p(w)$. There exists a string s such that we can choose $p_w = s\hat{w}p$.

Why? We use a computable prefix-free encoding $\hat{w}$ of w. The function

$$\psi_u(x) = \begin{cases} \varphi_p(w) & \text{if there exist } p \text{ and } w \text{ such that } u = \hat{w}p, \\ \text{undefined} & \text{otherwise.} \end{cases}$$

is partial computable and hence, $\psi_{\hat{w}p}(x) = \varphi_p(w)$; and this equals $\varphi_{s\hat{w}p}$ for some s.

In the next subsection we give one more explicit reference to the universality properties above, and afterwards, in the description of algorithms, we simply use instructions like "simulate the n-th Turing machine on input n" without reference.

6 Exercises

From now on, we present algorithms by high level descriptions: we assume that familiar mathematical functions on natural numbers are computable, we use instructions like "search for a string that satisfies the following properties", "sort the following sequence", etc. Let $\langle X \rangle$ stands for "some reasonable encoding[2] of X".

Exercise 11. A decider is *deterministic* if every state has at most one outgoing edge. Show that deterministic deciders recognize the same languages. Do you use breadth-first search or depth first search? Are both possible?

Exercise 12. Show that the following sets are decidable

$$\begin{aligned} A_{\text{NFA}} &= \{\langle B, w\rangle : B \text{ is an NFA that accepts } w\} \\ E_{\text{NFA}} &= \{\langle B\rangle : B \text{ is an NFA that accepts no strings}\} \\ EQ_{\text{NFA}} &= \{\langle B, C\rangle : B, C \text{ are NFAs that accepts the same strings}\}. \end{aligned}$$

[2] For some class of computation devices that decide a language, it means that one can compute a list $\langle L_1\rangle, \langle L_2\rangle, \ldots$ that contains all the computation devices of the class, such that the set

$$\{(n, w, t) \in \mathbb{N} \times \{0,1\}^* \times \mathbb{N} : \ L_n \text{ accepts } w \text{ after at most } t \text{ computation steps}\},$$

is computable.

Hint for the last one: use that regular languages are closed under complement, intersection and union.

Show also that one can test whether two *deterministic* automata with at most k states are equivalent by testing whether they are equivalent for all strings of at most $f(k)$. How large should f be? How large should f be for non-deterministic automata?

Exercise 13. Let A be the set of all descriptions $\langle M \rangle$ of nondeterministic automata M that accept a string w that contains a substring 111. Show that A is decidable.

Exercise 14. 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. Show that

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

is decidable.