Introduction to Computational Complexity

(Berechenbarkeit und Komplexität)

Johannes Kepler University Linz / Fall 2024

Univ.-Prof. Dr. Richard Kueng, MSc ETH

Date: Fall 2023

Contents

1. Motivation and (some) background

Date: October 3, 2024

1.1 Motivating example: traveling salesperson (TSP)

Agenda:

1. Motivating example: traveling salesperson
2. Overview of topics
3. Background: alphabets and strings

In theoretical computer science, we try to make absolute statements about computation. One of the core objectives – and a core focus of this lecture – is to determine whether a given computational problem is 'easy' or 'hard'. Some computational tasks, like finding a certain item within a list, or adding two natural numbers, are very 'easy' to solve on computing devices. Other computational problems, however, seem to be much, much 'harder'.

Computational Problem (traveling salesperson (TSP)). Given a map with n cities, find the *shortest* possible routes that visits all cities exactly once and terminates at the origin city.

We refer to Figure 1.1 for an illustration. Before moving on to discuss potential solutions, let us first see how a given TSP problem can be fully specified. A moment of reflection reveals that the actual map is not that important. What matters are pairwise distances between cities.

Fact 1.1 A given TSP instance involving n cities is completely specified by $\approx n^2$ pairwise city distances (in, say, kilometers). ∎

Exercise 1.2 For n cities, it is not necessary to store all possible n^2 pairwise distances, because there are redundancies (e.g. the distance Linz↔Wien is the same as Wien↔Linz). What is the minimal number of parameters (pairwise distances) that is required to completely specify a TSP instance?

Fact 1.1 asserts that a complete description of a given TSP problem scales *quadratically* in the number of cities n. Every TSP instance involving n cities

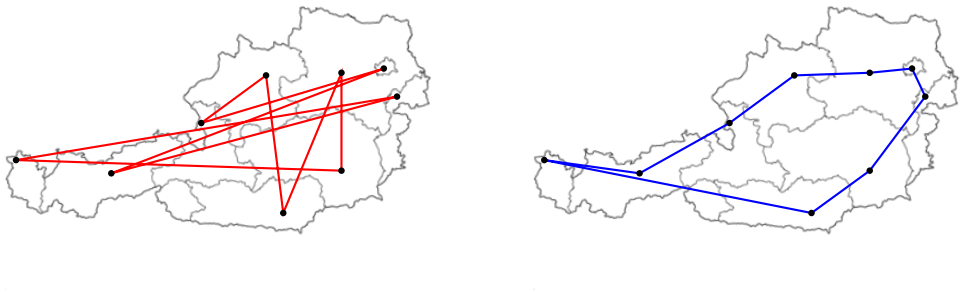


Figure 1.1 *Illustration of the traveling salesperson problem (TSP)*: A traveling salesperson intends to visit all Austrian state capitals (black dots) exactly once during a business trip. After the trip is completed, she also wants to return to the origin city. The goal is to find the shortest possible route. *(Left):* a very bad route, actually the worst possible route. *(Right):* a very good route, actually the best possible route. We leave it as an instructive exercise to verify these claims (e.g. via brute force search).

can be specified by roughly n^2 parameters. An Austrian TSP problem (9 state capitals), for instance, is fully characterized by 36 pairwise city distances (which is smaller than $9^2 = 81$). For Germany (16 state capitals), this number grows to is 120 (again smaller than $16^2 = 256$), while 1225 pairwise distances are required for the US (50 state capitals).

It is somewhat inconvenient that the description size of TSP grows faster than linear with the problem size (number of cities n). But, on the other hand, this growth is also not very extreme. Especially if we compare it to more explosive growth phenomena, like the one we are going to discuss next.

Now, that we have specified a given TSP problem, let us discuss how to solve it. We can specify each route by a list of $n + 1$ city names such that the first and last entry must be identical (the salesperson must return to the city of origin). For the Austrian TSP, Bregenz $\longrightarrow$ Eisenstadt $\longrightarrow$ Graz $\longrightarrow$ Innsbruck $\longrightarrow$ Klagenfurt $\longrightarrow$ Linz $\longrightarrow$ Salzburg $\longrightarrow$ St. Pölten $\longrightarrow$ Wien $\longrightarrow$ Bregenz (visit all Austrian state capitals in alphabetical order) is one possible route, albeit a very bad one.

Given access to pairwise city distances, it is easy to compute the total distance in kilometers: it requires exactly $(n + 1)$ additions (one for each leg of the route). That is, the cost of computing a route actually scales *linearly* with the number of cities n. This is very good, but keep in mind that we are tasked to find the shortest possible route. And a naive approach, like brute-force search, may require us to compare *very many* possible routes.

Fact 1.3 A given TSP instance involving n cities admits $\approx n! = n(n-1)(n-2)\cdots 1$ different routes. ∎

Exercise 1.4 Every permutation of n city names gives rise to a possible route (this is where the scaling with $n!$ comes from). But, do all these $n!$ reorderings lead to actually different routes? Determine the actual number of possible routes as a function of the number of cities.

The factorial of n quickly becomes enormous ($n!$ scales roughly like $n^n \approx \exp(n\log(n))$ which grows even faster than 2^n). And this does affect a brute-force search over all possible route distances. Suppose, for illustration, that we can compute the distance of a given route in 1ms = 10^{-3}s (and we can keep track of the smallest kilometer count seen so far at almost zero extra cost). Then, solving the Austrian TSP (9 state capitals) with brute-force search would require $\approx 9! \times 10^{-3} \approx$ 6min, which is still doable. But the German TSP (16 state capitals) would already require $16! \times 10^{-3}$s $\approx$ 663.5ys! For the US (50 state capitals), the rough number of different routes is $50! \times 10^{-3}$s $\approx 3 \times 10^{61}$s. This number is astronomically large in a very real sense. It is five orders of magnitudes larger than the number of atoms in the entire solar system (roughly 1.2×10^{56}). We cannot hope to compare all these route distances.

So, what is happening here? Is brute-force search for TSP merely a very bad algorithm design choice? Or is TSP perhaps an intrinsically difficult problem where the solution cost must, at least sometimes, scale very poorly with the input size (number of cities)? Of course it is possible to get much better solution strategies by actually looking at the map in question, or even implementing a smarter search procedure. But there may, actually, also be *fundamental limitations* to such improvements. Most computer scientists, myself included, believe that an exponential scaling in the number of cities n might be unavoidable in general. That is, even the best possible algorithm must sometimes require $\gtrsim \exp(cn)$ seconds, where $c > 0$ is some (unknown) constant. In words: there are scenarios, where solving TSP is *really expensive*. Throughout the course of this lecture, we will see why so many researchers believe that problems like TSP are intrinsically difficult.

At this point, it is worthwhile to emphasize that the above statement does not claim that all TSP problems are difficult. Certainly, there exist TSP problems that are very easy to solve. A concrete example from our real world is Asian Russia, where all noteworthy cities are arranged in a one-dimensional line – the Trans-Siberian Railway. Instead, the above claim states that there are, at least some, TSP problem instances that must be challenging for every solution strategy conceivable.

1.2 Overview of topics

This course will consist of, in total, 14 lectures. A tentative list of topics is as follows:

tentative list of topics

1. Motivation and (some) background
2. Finite state automata
3. Turing machines

4 Decision problems and languages
5 Universal Turing machines and undecidability
6 Time-bounded computations (**P**)
7 The problem class **NP**
8 Cook-Levin theorem and **NP** completeness
9 Karp reductions and some **NP** complete problems
10 Space complexity
11 **co-NP** and the polynomial hierarchy
12 Circuits
13 Circuit size-bounded computations
14 Circuit lower bounds & Circuit-SAT

computability

The first batch of lectures (lectures 2-5) is dedicated to fundamental questions about **computability**. We will first introduce an abstract model – the finite state automaton – that allows us to model simple computing devices. Subsequently, we will see that the addition of a working memory will make such simple computing devices much more powerful. The resulting Turing machine is so powerful, in fact, that it can simulate every possible computing device (think: universal compiler) and, therefore, can solve *a lot* of computing problems. Nonetheless, we will see that there are computing problems that even a Turing machine cannot solve. Such problems are *uncomputable*.

computational complexity

In the second batch of lectures (lectures 6 – 11), we are going to take a (comparatively) deep dive into **computational complexity theory**. Roughly speaking, this is the art of distinguishing between *easy problems* – in the sense that they can be computed efficiently – and *hard problems*. To achieve this goal, we will define different **problem classes** which can be used to group computational problems by difficulty. The problem class **P**, for instance, subsumes all computational tasks that we consider to be efficiently solvable. The problem class **NP**, on the other hand, subsumes all computational tasks where we can efficiently check whether a proposed solution is correct. We will study the relations and interdependencies between complexity classes. For instance, it is easy to see that every problem in **P** is also in **NP** (it is easy to check correctness of a solution to a problem that can itself be solved efficiently; simply solve the problem and compare). But it also seems reasonable to believe that the two classes are distinct (**P** $\neq$ **NP**): checking correctness of a solution should, after all, be easier than coming up with a solution ourselves. But, perhaps puzzingly, we do not (yet) have mathematical proof. The ***P*** $\neq$ **NP**-conjecture is one of the seven Millenium Size Problems in mathematics (should you be able to solve it, you'll get a 1 000 000USD prize and eternal fame). For context, the TSP problem from Section 1.1 is closely related to a problem in **NP**. And, because we believe **P** $\neq$ **NP**, we also believe that TSP should be a difficult problem.

circuit complexity

In the final batch of lectures (lectures 11 – 13), we will revisit **computational complexity from the perspective of logical circuits**. In the circuit picture, a computational problem is easy if we can solve it by evaluating a circuit that

is not too large (and not outlandishly complicated). It is hard if this is not possible in general. Although a well-established subfield in its own right, an early focus on circuit complexity theory is not standard for an introductory course in theoretical computer science. But, in order to truly understand a topic, it is often beneficial to approach it from multiple angles. Moreover, the circuit complexity picture plays into core strengths of the JKU curriculum (hardware design) and is also the point of departure for quantum computational complexity theory (which problems can be efficiently solved if we had access to a fully functional quantum computer; which problems would still remain challenging).

On first sight, theoretical computer science can seem like a cumbersome, old-fashioned research field with few actual implications. But this could not be further from the truth. Over the past decades, computing and, by extension, computer science has made a lasting impression on virtually all scientific disciplines. And theoretical computer science studies the fundamental possibilities and limitation of this toolbox. Results of this form have profound implications in a variety of scientific disciplines. We will use roughly half of our exercise classes to discuss such implications. Here is a tentative list of topics:

some modern implications of theoretical computer science

1. *swarm intelligence* (and cellular automata)
2. *computing power is everywhere* (guest lecture by Hel Pfeffer)
3. *what is glass?* (computational explanation for amorphous materials)
4. *quantum supremacy* (can quantum architectures beat existing hardware)
5. *a (very) brief introduction to quantum computers*

Since these topics will also be part of actual exercises, we will not cover them in this set of lecture notes.

1.3 Background: alphabets and binary encodings

The fundamental building blocks of information processing is a collection of symbols that we use for writing down things and/or performing computations.

Definition 1.5 (alphabet). An *alphabet* is a finite (and nonempty) set of symbols.

alphabets

We will typically use capital Greek letters, e.g. Σ, to denote a given alphabet. Our favorite alphabet will be the *binary alphabet* which consists of exactly two symbols: $\{0, 1\}$. Once we have agreed on an alphabet, we can combine the permitted symbols to represent more complicated expressions.

binary alphabet

Definition 1.6 (strings and length). A *string* over an alphabet is a finite sequence of symbols drawn from the alphabet. The *length* of a string is the total number of symbols it contains (including repetitions).

strings and length

Strings are very intuitive objects for computer scientists. We typically denote them by lower-case latin letters, e.g. x, and write $|x|$ for the length. For the binary alphabet $\{0, 1\}$, strings correspond to, well, bitstrings. For instance, $x = 1111110$ is a bitstring of length $|x| = 7$. Alphabets and, by extension,

strings can contain any number of symbols. Sometimes, they have an intuitive meaning, but this is not necessary. Different choices, such as

$$\underbrace{\{A, B, C, \ldots, Z\}}_{\text{latin alphabet, capital letters}} \quad \text{and} \quad \underbrace{\{0, 1, \ldots, 9\}}_{\text{decimal digits}}$$

are both equally valid alphabets (albeit with a different number of symbols). In fact, the actual symbols that appear within a given alphabet do not really matter all that much. We can immediately forget about a given alphabet and, instead, represent the symbols by bitstrings. This is the content of the following mathematical statement.

bit encoding

Theorem 1.7 (bit encodings). Every symbol from a given alphabet can be represented as a bitstring. This representation is one-to-one and only scales logarithmically in alphabet size.

It is worthwhile to dissect the actual meaning of this succinct formal statement. Suppose that we have an alphabet Σ with N different symbols. We can denote different symbols by $a_0, \ldots, a_{N-1}$ (remember that the actual form of symbols does not matter). Theorem 1.7 asserts that we can replace each symbol a_i with a bitstring x_i (where $0 \leq i \leq N-1$). *One-to-one* means that the identification $a_i \leftrightarrow x_i$ is unique: each x_i represents exactly one a_i and vice versa. The final part of the statement asserts that these bitstrings need not be very long: $|x_i| \approx \log_2(N)$ for all $1 \leq i \leq N$.

Exercise 1.8 Prove Theorem 1.7.
Hint: It is helpful to first replace original alphabet symbols by integers ranging from 0 to $N-1$ (convince yourself that this replacement is one-to-one) and take it from there.

Bit encodings are ubiquitous in computer science (i.e. they occur everywhere). Concrete examples are *binary encodings of natural numbers*,e.g. $7 \leftrightarrow 0111$, or *ASCII encodings* of characters, e.g. ? $\leftrightarrow$ 011 1111.

Idea: Throughout this course, we will make liberal use of the bit encoding theorem. Whenever possible, we will try to formulate models & concept in terms of the binary alphabet $\{0, 1\}$ and bitstrings. Not only is this the 'language' computers actually speak, but it will also liberate us from overly cumbersome exposition.

Problems

Problem 1.9 (TSP: parameter counting). How many parameters (pairwise distances) are required to fully specify a TSP problem involving n cities? What is the total number of inequivalent routes?

Numerics 1.10 (TSP for Austrian state capitals). Solve TSP for all 9 Austrian state capitals. Use map services, like Google maps, to extract all (relevant) pairwise

distances between Austrian state capitals. Then, write a piece of software (in the programming language of your choice) that computes all possible route distance and subsequently compares them. What is the best possible route to visit all 9 state capitals? And what is the worst?

Problem 1.11 (bit encoding). Prove the bit encoding theorem presented in Theorem 1.7.

2. Finite state automata

Date: October 10, 2024

In this lecture we introduce one of the most basic computational models – the *finite state automaton*. By itself, it is not particularly powerful. But, it lends itself to a clean and self-contained analysis. More expressive computational models, like the *Turing machine model*, build upon these core ideas. Also, when many finite state automata act together in unison, they can achieve truly amazing things. This will be the topic of our first special lecture on 'swarm intelligence' (not included in these lecture notes).

Agenda:

1. Motivating examples:
   (i) automatic door
   (ii) computing parity
2. Deterministic finite automata (DFAs)
   (i) formal definition
   (ii) computations
3. Nondeterministic finite automata (NFAs)
   (i) determinism vs. nondeterminism
   (ii) equivalence relations (without proof)

2.1 Motivating examples

2.1.1 Motivating example: automatic door

As we shall see later on, finite state automatons are abstract models of computing devices that have extremely limited memory. On first sight, a tiny memory may

Figure 2.1 *Setup for an automatic door (top view):* the door in question does not open sideways (like most automatic doors in shopping centers), but towards the rear (like the entrance door to the Open Innovation Center on JKU campus).

seem extremely restricting. But there are many useful machines that get by with it. In fact, we interact with such devices all the time. The controller for an automatic door is one such example. And we interact with it every time we go shopping. So let's give it the attention it may deserve[1]. To make things a bit more interesting, we consider an automatic door that opens towards the rear and not the sides, see Figure 2.1

The job description of such an automatic door is simple: If a person approaches, open the door long enough to let them pass and then close it again. In order to do its job, the automatic door must be able to sense if a person approaches from the front. And, towards the rear, it must also be able to sense the presence of a person. Because opening/closing the door is only advisable if the rear area is empty. These requirements can be achieved by two pads – one in the front and one in the rear – that detect whether a person is standing in either of the two critical locations (the blue and red squares in Figure 2.1). So, for all practical purposes, our door machine lives in a world where there only exist four possible 'inputs' at any time:

$$\Sigma = \{\text{'neither'}, \text{'only front'}, \text{'only rear'}, \text{'both'}\}\,.$$

This is, of course, a valid alphabet. And nothing prevents us from combining symbols from that alphabet into strings, e.g.

$$x = \underbrace{\text{'neither'} \cdots \text{'neither'}}_{9 \text{ times}} \text{ 'only front' 'only rear' } \underbrace{\text{'neither'} \cdots \text{'neither'}}_{9 \text{ times}}$$

is a string of length $|x| = 20$ over the alphabet Σ. We can actually interpret this string in the original context of an automatic door: x describes 20 frames of a (boring) movie: a person approaches an automatic door, passes through it, and leaves again. Starting from the left, we see that the initial configuration is $x[0] =$ 'neither'. That is, no person is close to the automatic door. And this situation remains unchanged for 9 time steps, i.e. $x[t] =$ 'neither' for $t = 0, \ldots, 8$. At time $t = 9$, finally something interesting happens: $x[9] =$ 'only front'. Somebody is approaching the automatic door from the front. Since $x[10] =$ 'only rear', we can conclude that it took the person one time step to enter the building. Afterwards, the person has crossed and the automatic door environment remains empty for another 9 time steps: $x[t] =$ 'neither' for $t = 11, \ldots, 19$.

Let us now check how our automatic door machine reacts to this movement pattern under the extra assumption that the door is initially closed. Let $q[t] \in \{\text{'CLOSED'}, \text{'OPEN'}\}$ denote the *state* of the automatic door at time t. Then, by assumption $q[0] =$ 'CLOSED'. Also, $x[0] =$ 'neither' (the person is still far away), so there is no reason for the automatic door to open at time $t = 0$. This configuration remains unchanged for the next 8 time steps. But, at

[1] This instructive motivating example is taken from Sipser's textbook *Introduction to the Theory of Computation [Sip97]*.

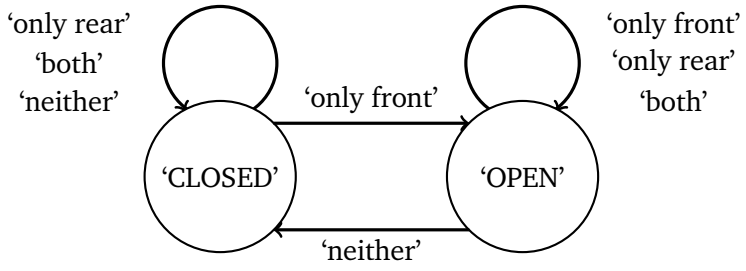


Figure 2.2 *State diagram describing an automatic door controller:* circles denote different states the door can be in ('OPEN' or 'CLOSED'). Possible transitions between states are depicted by directed arrows. E.g. the trigger 'only front' makes the door change states from 'CLOSED' to 'OPEN'.

	'neither' (00)	'only front' (01)	'only rear' (10)	'both' (11)
'CLOSED'	'CLOSED'	'OPEN'	'CLOSED'	'CLOSED'
'OPEN'	'CLOSED'	'OPEN'	'OPEN'	'OPEN'

Table 2.1 *Transition table describing an automatic door controller.*

time step $t = 9$, the person has approached and is now standing on the front pad with the intention to enter. The door sensor recognizes the configuration 'only front' which prompts the automatic door to change its state to 'OPEN', as it should. This allows the person to enter and at time step $t = 10$, she is already on top of the rear pad. The door recognizes the configuration 'only rear' which stops it from closing (if the door closed now, there would be a collision). But, at $t = 11$, the passing person has left the rear area and the door is free to close again. This configuration also doesn't change anymore for the remaining time steps. In summary, the door configurations should be

$$\begin{aligned} q[0] &= \text{'CLOSED'}, \ldots, q[9] = \text{'CLOSED'}, \\ q[10] &= \text{'OPEN'}, q[11] = \text{'OPEN'}, \\ q[12] &= \text{'CLOSED'}, \ldots, q[19] = \text{'CLOSED'}. \end{aligned}$$

Note that the time evolution of our automatic door would only slightly change if we started in configuration $q[0] =$ 'OPEN'. Since the first pad configuration is 'neither', the door would simply close and remain so until our passenger approaches at time $t = 9$. There are two appealing ways to concisely summarize the inner workings of our automatic door:

(i) a visual illustration in terms of a *state diagram* presented in Figure 2.2;
(ii) a *transition table* presented in Table 2.1.

We will discuss both representations in detail later on.

For now, we take inspiration from this description by interpreting the reaction of the automatic door to external 'person configurations' as a certain type of 'computation'. And building up on this idea will pilot us into more interesting territory. At this point, let us emphasize that our automatic door computer has a fatal flow: the 'computation' never stops. Our automatic door machine requires a constant stream of new inputs from the alphabet and reacts accordingly. Of course, we want an automatic door to behave exactly like that. But for a computing device this is dangerous. Programs that never halt are a scary thing!

2.1.2 Closer to actual computation: a parity checking machine

We have seen that we can interpret very simple mechanical devices as certain types of computing architectures. But this correspondence can seem a bit artificial. Let us now present another simple example that is closer to digital computation. The machine, we envision, is designed to process bits. I.e. it works on our favorite alphabet $\Sigma = \{0, 1\}$. And, similar to the automatic door discussed above, it can be in one of two *states*. This time, we are a bit more fancy (and less creative) and denote them by q_0 and q_1, respectively. Such pristine mathematical notation should remind us that the actual label we assign to these states does not matter. And this time, we also resolve two weaknesses we encountered when discussing the automatic door:

(i) We should fix an *initial state* (to resolve ambiguities at the beginning of the computation). Let us pick q_0 as *starting state*.

(ii) We should also identify a *final state* that allows the computation to output a result. Let us pick q_1 as *accept state*.

Of course, we must also specify the actual workings of our machine. Since it is intended to process bits ($\Sigma = \{0, 1\}$) and possesses only two possible states, a 2×2 *transition table* suffices. We choose

	0	1
q_0	q_0	q_1
q_1	q_0	q_1

,

and can, equivalently, represent the entire machine by a diagram, see Figure 2.3. Let us denote our freshly assembled computing machine by M. After all, we haven't explored yet what it actually does. To find that out, we have to play around a bit. Our machine starts in the state q_0 and is hard-wired to process individual bits in a certain fashion. Hence, we can use it to process bitstrings from left to right. The machine *accepts* a bitstring if, after the entire string is processed, it ends up in the accept state q_1. Otherwise it *rejects* the string. We can use the state diagram to conveniently cycle through the first couple of bitstrings. The notation might seem a bit cumbersome, but should explain itself from context. (Recall, that we are computer scientists and start counting with 0. In particular, $x[0]$ denotes the first element of a bitstring (from the left)):

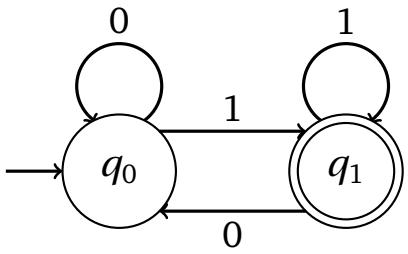


Figure 2.3 *State diagram for a simple machine that processes bitstrings:* The two possible internal states are denoted by circles, directed arrows between the two label transitions. The single arrow coming from the left marks the starting state (q_0 in this case). The 'double circle' singles out final states that allows the computation to terminate and 'accept' (q_1 in this case).

$$
\begin{aligned}
x = 0, |x| = 1: &\quad q_0 \xrightarrow{x[0]=0} q_0 \Rightarrow \text{reject},\\
x = 1, |x| = 1: &\quad q_0 \xrightarrow{x[0]=1} q_1 \Rightarrow \text{accept},\\
x = 00, |x| = 2: &\quad q_0 \xrightarrow{x[0]=0} q_0 \xrightarrow{x[1]=0} q_0 \Rightarrow \text{reject},\\
x = 01, |x| = 2: &\quad q_0 \xrightarrow{x[0]=0} q_0 \xrightarrow{x[1]=1} q_1 \Rightarrow \text{accept},\\
x = 10, |x| = 2: &\quad q_0 \xrightarrow{x[0]=1} q_1 \xrightarrow{x[1]=0} q_0 \Rightarrow \text{reject},\\
x = 11, |x| = 2: &\quad q_0 \xrightarrow{x[0]=1} q_1 \xrightarrow{x[1]=1} q_1 \Rightarrow \text{accept}.
\end{aligned}
$$

Now, we can see that a pattern emerges. Our machine seems to accept precisely those strings that end with a 1. A couple of test calculations with longer bitstrings (which we won't do here) confirm this intuition. This pattern is far from arbitrary. And it gains additional meaning when we view the bitstrings to be processed as bit encodings $\llcorner n \lrcorner$ of natural numbers $n \in \mathbb{N}$: $\llcorner 0 \lrcorner = 0, \llcorner 1 \lrcorner = 1, \llcorner 2 \lrcorner = 10, \llcorner 3 \lrcorner = 11$, etc. Let us write $M(n) = 1$ if our machine M accepts the bit encoding $\llcorner n \lrcorner$ of n. And if it rejects the string, we write $M(n) = 0$. Then, the pattern we identified translates to

$$M(0) = 0,\ M(1) = 1,\ M(2) = 0,\ M(3) = 1,\ M(4) = 0.$$

This pattern generalizes and we can, in fact, conclude

$$M(n) = \begin{cases} 1 & \text{if } n \in \mathbb{N} \text{ is an odd number,} \\ 0 & \text{else if } n \in \mathbb{N} \text{ is an even number.} \end{cases}$$

Our simple machine M accepts precisely those bitstrings that encode odd natural numbers. And it rejects bit encodings of even numbers. In mathematics, the function that computes whether an integer number is odd or even is called

parity. So, our machine computes the parity of any natural number. This is pretty cool, because the parity is also a very useful function in computer science. And we only needed a machine with two internal states to compute it. In fact, this machine is even less complicated than an automatic door! parity

Our device, however, is not limited to process bitstrings of length two. It can compute the parity of arbitrary natural numbers. The *runtime* (number of steps) it requires to do that is equal to the length of the bit encoding. This is a very fast operation, because the length of bit encodings $\llcorner n \lrcorner$ only scales logarithmically in the size of the actual number:

$$\text{runtime}_M(n) = |\llcorner n \lrcorner| = \lfloor \log_2(n) \rfloor + 1. \tag{2.1}$$

The logarithm is one of the most slowly growing functions we know. So, computing the parity remains tractable even for very, very large numbers[2]. Additional improvements are possible if we instead compute the parity using a decimal alphabet, or even a hexadecimal alphabet.

Exercise 2.1 (decimal parity function). Design a machine that computes parity by processing decimal numbers instead of binary ones. Show that the resulting runtime is comparable to, but slightly faster than, the runtime of a binary parity checking machine.

We can use similar ideas to construct machines that execute other important functionalities.

Exercise 2.2 (parity of sums). Construct a machine that computes the parity of a sum of bits, i.e. $M(x_0 \cdots x_{n-1}) = \text{parity}(x_0 + x_1 + \cdots + x_{n-1})$ for any bitstring length $n \in \mathbb{N}$. If we restrict attention to bitstrings $x_0 x_1$ of length $n = 2$, then this machine computes a very prominent logical function (gate). Which one is it?

We see that several fundamental primitives in computation and hardware design seem to correspond to simple machines that check bits one at a time and change their internal state based on a pre-specified set of transition rules. This is not a coincidence.

2.2 Deterministic finite automata (DFAs)

2.2.1 Formal definition

The two examples above motivate the first model of computation we will discuss in this course. It is very simple, not particularly powerful, and called the *(deterministic) finite automata* model. Deterministic finite automate are also called *finite state machines*.

Definition 2.3 (deterministic finite automaton (DFA)). A *deterministic finite automaton (DFA)* is a machine that can either accept or reject strings by sequentially processing symbols (from left to right). It is fully characterized by

[2]The (base-2) logarithm of our favorite astronomically large number – 2.4×10^{67}, aka the number of atoms in the milky way galaxy – is approximately 224.

- a (finite and nonempty) set of internal *states* Q;
- the alphabet Σ of symbols it can process;
- a *transition function* $\delta : Q \times \Sigma \to Q$ that describes the inner working;
- a designated *start state* $q_0 \in Q$ and;
- a subset of *accept states* $F \subseteq Q$.

Formally, we identify a DFA with the *5-tuple* $M = (Q, \Sigma, \delta, q_0, F)$. formal DFA definition as 5-tuple

We already have seen a concrete example. The parity check machine from Section 2.1.2 can be succinctly characterized as

$$M = \Big(\underbrace{\{q_0, q_1\}}_{Q}, \underbrace{\{0,1\}}_{\Sigma}, \delta, q_0, \underbrace{\{q_1\}}_{F}\Big),$$

where the transition function $\delta : \{q_0, q_1\} \times \{0,1\} \to \{q_0, q_1\}$ acts as

$$\delta(q_0, 0) = q_0, \;\; \delta(q_0, 1) = q_1, \;\; \delta(q_1, 0) = q_0, \;\; \delta(q_1, 1) = 1.$$

Note that this is simply another way of writing down the information stored in the transition table. It is common to express DFAs visually by using *state diagrams*. We have already seen two examples in Figure 2.1 and Figure 2.3, respectively. Here are the general rules: state diagrams

1. States are denoted by circles.
2. These circles are connected by (directed) arrows. And alphabet symbols label these arrows.
3. The transition function is characterized by the arrows, their labels and the circles they connect.
4. The start state is determined by the arrow coming in from nowhere.
5. Accept states are highlighted by double circles.

Figure 2.4 depicts a somewhat more involved state diagram that is taken from [Wat20]. Let us try to convert it into more formal language. The alphabet is $\Sigma = \{0,1\}$, because arrows are either labeled by 0 or 1. Also, there are six internal states labeled by q_i $(0 \le i \le 5)$, so $Q = \{q_0, q_1, q_2, q_3, q_4, q_5\}$. The starting state is q_0 and there are three accept states: $F = \{q_0, q_2, q_5\}$. Finally, the transition function $\delta : Q \times \Sigma \to Q$ acts as follows: transition function

$$\begin{aligned}
&\delta(q_0,0) = q_0, \quad \delta(q_0,1) = q_1,\\
&\delta(q_1,0) = q_3, \quad \delta(q_1,1) = q_2,\\
&\delta(q_2,0) = q_5, \quad \delta(q_2,1) = q_5,\\
&\delta(q_3,0) = q_3, \quad \delta(q_3,1) = q_3,\\
&\delta(q_4,0) = q_4, \quad \delta(q_4,1) = q_1,\\
&\delta(q_5,0) = q_4, \quad \delta(q_5,1) = q_2.
\end{aligned} \tag{2.2}$$

See also Table 2.2 for the corresponding transition table. To summarize, the DFA described in Figure 2.4 corresponds to the 5-tuple

$$M = \Big(\underbrace{\{q_0, q_1, q_2, q_3, q_4, q_5\}}_{Q}, \underbrace{\{0,1\}}_{\Sigma}, q_0, \delta, \underbrace{\{q_0, q_2, q_5\}}_{F}\Big),$$

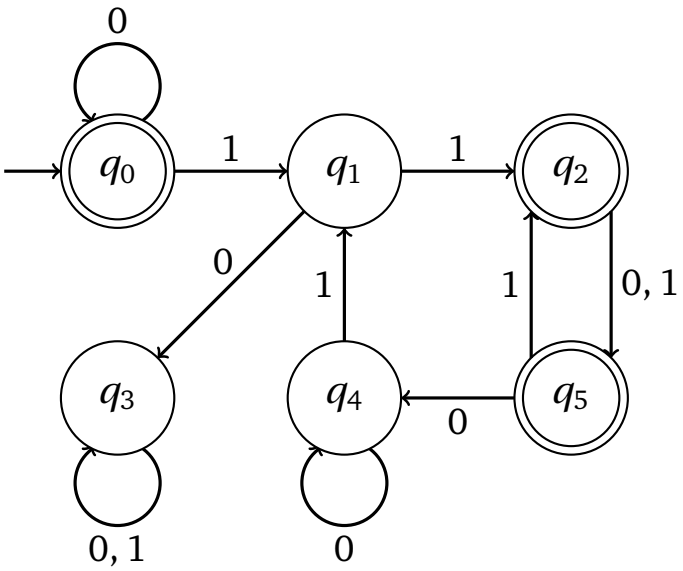


Figure 2.4 *Example of a more involved DFA state diagram.*

where the transition function is characterized by Eq. (2.2).

Exercise 2.4 Which of the following bitstrings are accepted by the DFA visualized in Figure 2.4: a) 00000010, b) 11101110, c) 11001100, d) 11010101.

Note that Figure 2.4 contains an arrow that is labeled by two alphabet symbols ('0, 1'). This means that there are actually multiple arrows, each labeled by a single symbol. Summarizing several single-symbol arrows with the same start and end location by a single arrow with multiple symbols declutters presentation and makes the entire diagram easier to read. Being able to read state diagrams can come in handy throughout various stages of computer science studies (and career). One way to make sure that one actually understands them, is to convert them into the formal language from Definition 2.3. Of course, you can also go the other way and draw a state diagram from a formal description of a 5-tuple. The problem section at the end of this chapter contains one problem for each direction.

2.2.2 DFA computations

We have already seen that DFAs can be used to perform computations. In particular, we showcased how a simple DFA – the parity check machine from Section 2.1.2 – *accepted* bit encodings of odd numbers and *rejected* bit encodings of even numbers. Thinking in terms of state diagrams makes it easy to say in words what that actually means. We begin at the start state and iteratively transition from one state to another based on the symbols of the input string (starting on the left and iteratively processing to the right). We accept if and only if we end up on an accept state. Otherwise we reject.

accepting and rejecting inputs

	0	1
q_0	q_0	q_1
q_1	q_3	q_2
q_2	q_5	q_5
q_3	q_3	q_3
q_4	q_4	q_1
q_5	q_4	q_2

Table 2.2 *Transition table for the DFA introduced in Figure 2.4.* Rows label different states ($q_0, \ldots, q_5$) while columns label different binary inputs (0 or 1).

This all makes sense intuitively, but it is not yet a formal definition. How do we define in precise, mathematical terms what it means for a DFA to accept or reject a string? In particular, intuitive guidelines, like 'follow transitions' and 'end up on an accept state' should be replaced by more precise mathematical statements. There is more than one way to achieve this goal. The following formal definition is taken from Watrous' lecture notes [Wat20].

Definition 2.5 (DFA computations). Let $M = (Q, \Sigma, \delta, q_0, F)$ be a DFA and let $x = x_0 \cdots x_{n-1}$ be a length-n string ($n \geq 1$) over the alphabet Σ. We say that the DFM M *accepts* x if there exist states $r_0, \ldots, r_n \in Q$ such that

$$r_0 = q_0, \quad r_{k+1} = \delta(r_k, x_k) \text{ for } k = 0, \ldots, n-1 \quad \text{and} \quad r_n \in F. \tag{2.3}$$

The DFA also accepts the empty string $x = \varepsilon$ (not a single symbol, $n = 0$) if $q_0 \in F$. If M doesn't accept x, then we say that M *rejects* x.

Note that the definition addresses the special case $x = \epsilon$ separately. This is important, because there are often multiple ways to deal with 'nothing', in particular empty strings, or empty sets. But often, there is only one way that leads to consistent behavior across all possibilities. And while it may seem overly pedantic to deal with the empty string at all, these things can start to matter if we want to combine simple formal statements (like this one) to obtain more interesting, and typically more involved, statements.

For nonempty strings ($n \geq 1$), the formal definition of acceptance is that there must exist a sequence of states $r_0, \ldots, r_n$ such that the first state is the start state, the last state is an accept state and each state in the sequence is determined from the previous state and the corresponding symbol read from the input as the transition function dictates. If we are in the state r_k and read the symbol x_k, then the new state must be $r_{k+1} = \delta(r_k, x_k)$. In other words: the entire computation must be correct (and deterministic, as we shall discuss next).

2.3 Nondeterministic finite automata (NFAs)

2.3.1 Determinism vs. nondeterminism

Note that the transition rules of a DFA follow stringent, yet intuitive, rules. They become apparent if we study DFA state diagrams, like the ones presented in Figure 2.3 and Figure 2.4, a bit more closely.

determininistic computations

First and foremost, every DFA state (circle) has exactly one exiting transition arrow for each symbol of the alphabet. And secondly, arrows can only be labeled by symbols from the alphabet Σ in question. These properties ensure that the machine is perfectly predictable. If we know the internal state r_k at time step k and the current alphabet symbol x_k to be read, then we know with certainty that the machine will transition into state $r_{k+1} = \delta(r_k, x_k)$. The philosophical view that events are determined completely by previously existing causes is called *determinism*. This is why we call these machines deterministic finite state automatas. The opposite of determinism is some kind of *nondeterminism* or randomness.

Determinism vs. nondeterminism has long been, and still is, an important conceptual debate in various scientific disciplines. Physics is a good example. Newtonian mechanics, for instance, is completely deterministic (e.g. if you know the current location of an asteroid, as well as its momentum, Newton's equations of motion allow you, in principle, to perfectly compute its future trajectory) and so is (special and general) relativity, as well as electromagnetism. Quantum mechanics, by contrast, is a probabilistic theory and therefore *not* deterministic. In the first half of the 20th century, this inherent randomness has worried some of the greatest scientific minds. Einstein's famous quote

> "*I, at any rate, am convinced that [God] does not throw dice*"
> (German: Jedenfalls bin ich überzeugt, daß der nicht würfelt.)

from 1926 (in a letter to Max Born, one of the fathers of quantum mechanics) is a testimony of such a spirited debate.

2.3.2 Nondeterministic finite state automata (NFAs)

The determinism vs. nondeterminism debate is also very important for computer science. In the remainder of this chapter, we briefly discuss it in the context of finite state machines. But, we will also see certain aspects of this question in later chapters of the course.

nondeterministic computations

State diagrams describing a *nondeterministic finite automaton (NFA)* can break the rules of DFAs in two ways:

(i) For each state (circle) and alphabet symbol, there can be zero, one, or more than one exiting transition arrow (state diagrams describing a DFA always have exactly one). Multiple arrows provide the NFA with the freedom of choice: it can choose which arrow to follow. The absence of an arrow instead prevents the computation from proceeding.

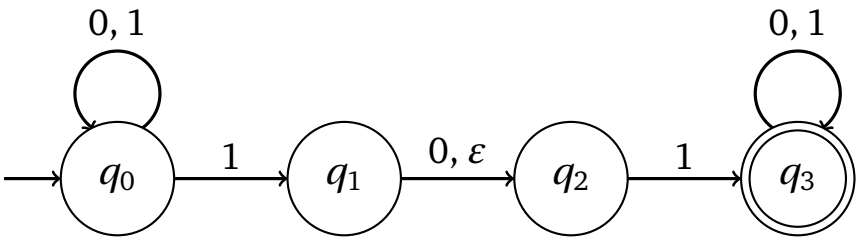


Figure 2.5 *State diagram of a NFA:* this state diagram violates the rules of a DFA. In particular, there are two 1-arrows at state q_0 (leftmost circle). The state q_1 has one exiting arrow for 0, but note for 1. The ε-arrow (empty string) between q_1 and q_2 indicates that such a transition is possible without reading a symbol from the string in question.

(ii) We introduce a new type of arrow, labeled by ε (or symbol for the empty string). This arrow enables the NFA to change its internal state without reading a string symbol.

Figure 2.5 describes a possible state diagram. Nondeterminism may be viewed as a kind of parallel computation. Multiple independent 'threads' can be executed concurrently. And, if at least one of these subthreads accepts, then the entire computation accepts.

Exercise 2.6 Consider the NFA from Figure 2.5 that performs computations over the binary alphabet $\Sigma = \{0, 1\}$. Which of the following strings is *not* accepted: a) 11, b) 101, c) 010, d) 11011 (can you recognize a pattern?)

We can also think of NFAs in terms of randomness. Each time, the automaton faces a situation with multiple ways to proceed (arrows to follow), it picks one uniformly at random. Conversely, it is also possible to end up at a state where there is no way of moving forward. Consider, for instance, the NFA from Figure 2.5: if we are in state q_1 and read in a 1, we cannot proceed. In this case, this particular thread 'dies' and is discarded.

We say that a NFA *accepts* a state if the probability of ending up in an accept state is strictly larger than zero. Note, in particular, that it is perfectly fine if this probability to accept is positive, but astronomically small. E.g. a numerical value of $1/(2.4 \times 10^{67})$ – one over the number of atoms in the milky way galaxy – still leads us to say that the NFA accepts a given string. This is in stark contrast to models of randomized computation, where we require accept probabilities to be strictly larger than $1/2$ (which can then be boosted arbitrary close to one by running the computation multiple times and taking a majority vote). Nondeterminism is really a statement about *possibilities*, not *probabilities*.

acceptance for NFAs

2.3.3 Equivalence between NFAs and DFAs

It is not entirely wrong to view NFAs as 'untrustworthy' DFAs that are allowed to cheat sometimes. And, on first sight, this ability should make them more

powerful at accepting complicated bitstring configurations. Perhaps surprisingly, this is *not* really the case.

NFAs and DFs are equivalent

Theorem 2.7 (equivalence between NFAs and DFAs). All computations performed by a NFA can be perfectly reproduced by a deterministic finite automaton DFA. However, there may be an exponential overhead in the number of states required.

This is a standard result in the theory of finite automata. Most introductory courses in theoretical computer science dedicate a considerable amount of attention to properly discuss and prove this statement. We, however, have neither the time, nor the proper background, to do that. Instead we refer the interested reader to standard textbooks [Sip97] or earlier versions of this course [Sch20] for further reading. Instead, we quickly elaborate on the content of Theorem 2.7 in words: Everything that can be done with a NFA can also be done with a DFA. This is very interesting from a philosophical point of view. Nondeterminism, which is stronger than performing computation with the help of additional randomness, does not increase the expressive power of finite automata at all. It can, however, be very expensive to build a DFA that does the same job as a NFA. The number of states required to make it work can, and often does, scale exponentially in the original number of NFA-states: $|Q_{\text{DFA}}| \approx 2^{|Q_{\text{NFA}}|}$. This is a consequence of the mathematical proof behind Theorem 2.7. The key idea is as follows: for a NFA that contains N states $Q = \{q_0, \ldots, q_{N-1}\}$, we construct a DFA with (up to) 2^N states, each of which describes a subset of NFA states, e.g. the (sub-)set $\{q_0, q_{N-1}\} \subseteq Q$ would be one state of the DFA to be constructed.

Problems

Problem 2.8 (decimal parity, see also Exercise 2.1). The aim is to design a DFA that computes parity by processing decimal numbers instead of binary ones. Only two states will be required. Let's call them q_{even} and q_{odd}.

1. Draw a state diagram that characterizes this DFA (it should contain two circles and 10 outgoing arrows for each of them; do use multiple labels to declutter presentation).
2. Convert this state diagram into a formal description according to Definition 2.3. Specify the transition function by completing the following table and inserting the correct states:

$$\begin{array}{cccccc} \delta(q_{\text{even}}, 0) = ?, & \delta(q_{\text{even}}, 1) = ?, & \cdots & \delta(q_{\text{even}}, 8) = ?, & \delta(q_{\text{even}}, 9) = ?, \\ \delta(q_{\text{odd}}, 0) = ?, & \delta(q_{\text{odd}}, 1) = ?, & \cdots & \delta(q_{\text{odd}}, 8) = ?, & \delta(q_{\text{odd}}, 9) = ?. \end{array}$$

   Hint: Formally, this is a table with 10 rows and 2 columns, but there is underlying structure that can be exploited.
3. Which of the following statements is correct:

a) runtime$(n) = \lceil \log_{10}(n) + 1 \rceil \approx \lceil 3.32 \times \log_2(n) + 1 \rceil$ which is *faster* than the runtime for computing binary parity, see Eq. (2.1).
b) runtime$(n) = \lceil \log_{10}(n) + 1 \rceil \approx \lceil 3.32 \times \log_2(n) + 1 \rceil$ which is *slower* than the runtime for computing binary parity, see Eq. (2.1).
c) runtime$(n) = \lceil \log_{10}(n) + 1 \rceil \approx \lceil 0.3 \times \log_2(n) + 1 \rceil$ which is *faster* than the runtime for computing binary parity, see Eq. (2.1).
d) runtime$(n) = \lceil \log_{10}(n) + 1 \rceil \approx \lceil 0.3 \times \log_2(n) + 1 \rceil$ which is *slower* than the runtime for computing binary parity, see Eq. (2.1).

Hint: Remember (or look up) how to change the basis of logarithms.

Problem 2.9 (parity of sums, see also Exercise 2.2). The aim is to construct a DFA that computes the parity of a sum of bits, i.e. $M(x_1 \cdots x_n) = \text{parity}(x_1 + x_2 + \cdots x_n)$ for any bitstring length $n \in \mathbb{N}$. Only two states will be required. Let's call them q_{even} and q_{odd}.

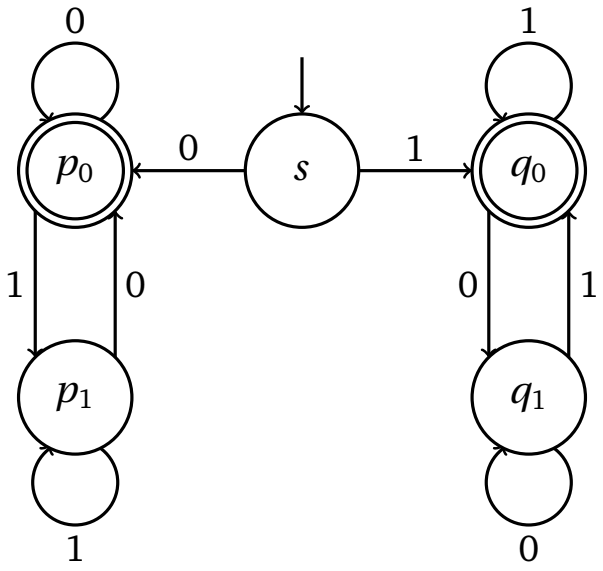
1 Draw a state diagram that characterizes this DFA (it should contain two circles and 2 outgoing arrows for each of them).
2 Convert this state diagram into a formal description according to Definition 2.3. Specify the transition function by completing the following table and inserting the correct states:

$$\delta\,(q_{\text{even}}, 0) = ?, \delta\,(q_{\text{odd}}, 1) = ?,$$
$$\delta\,(q_{\text{odd}}, 0) = ?, \delta\,(q_{\text{even}}, 1) = ?.$$

3 If we restrict attention to length-2 bitstrings $x_1 x_2$, then this DFA computes a very prominent logical function (gate). Which one is it?

a) AND $(x_1 \wedge x_2)$
b) NAND $(\neg(x_1 \wedge x_2)$,
c) OR $(x_1 \vee x_2)$,
d) XOR $((x_1 \vee x_2) \wedge \neg(x_1 \wedge x_2))$.

Problem 2.10 (state diagram to 5-tuple). Consider the following state diagram of a DFA working over the binary alphabet $\{0, 1\}$:



1 Convert this state diagram into a formal description according to Definition 2.3 (5-tuple). Specify the transition function by replacing the question marks in the following table by the correct states:

$$\begin{array}{ll} \delta(s,0) =? & \delta(s,1) =?, \\ \delta(p_0,0) =?, & \delta(p_0,1) =?, \\ \delta(p_1,0) =?, & \delta(p_1,1) =?, \\ \delta(q_0,0) =?, & \delta(q_0,1) =?, \\ \delta(q_1,0) =?, & \delta(q_1,1) =?. \end{array}$$

2 Which one of the following statements is true:

a) This DFA accepts all bitstrings that start and end with a different symbol.
b) This DFA accepts all bitstrings that start and end with the same symbol.
c) This DFA accepts all bitstrings that start with a 1.
d) This DFA accepts all bitstrings that end with a 1.

Problem 2.11 (5-tuple to state diagram). Consider the DFA defined by the 5-tuple $M = (\{q_0, q_1, q_2\}, \{0,1\}, \delta, q_0, \{q_2\})$, where the transition function δ acts like

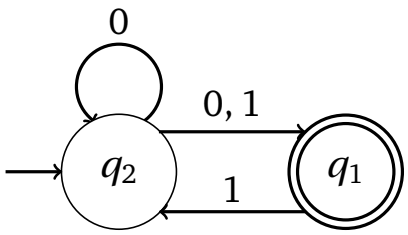
$$\begin{array}{ll} \delta(q_0,0) = q_0, & \delta(q_0,1) = q_1, \\ \delta(q_1,0) = q_2, & \delta(q_1,1) = q_1, \\ \delta(q_2,0) = q_1, & \delta(q_2,1) = q_1. \end{array}$$

1 Draw the associated state diagram.
2 Which one of the following statements is true:

a) This DFA accepts all bitstrings that contain at least one 1 and an even number of 0s that follow the last 1.
b) This DFA accepts all bitstrings that contain at least one 0 and an even number of 1s that follow the last 0.
c) This DFA accepts all bitstrings that contain at least one 1 and an odd number of 0s that follow the last 1.
d) This DFA accepts all bitstrings that contain at least one 0 and end with '10' following the last 0.

Problem 2.12 Which of the following bitstrings are accepted by the DFA visualized in Figure 2.4: a) 00000010, b) 11101110, c) 11001100, d) 11010101.

Problem 2.13 (NFA-to-DFA conversion (challenging)). Consider the NFA over the binary alphabet $\{0,1\}$ described by the following state diagram:



Construct a state diagram that describes a DFA that perfectly reproduces its functionality.

Hint: there are four subsets of the set $\{q_1, q_2\}$ of NFA states: $\emptyset$ (the 'empty set'), $\{q_1\}$ ('only q_1'), $\{q_2\}$ ('only q_2') and $\{q_1, q_2\}$ ('both q_1 and q_2'). The DFA in question works on the binary alphabet $\{0, 1\}$, has 4 states in total (one for each subset) and 8 transition arrows (one outgoing 0-arrow and one outgoing 1-arrow for each state).

3. Turing machines

Date: October 17, 2024

Last time we have introduced finite state automata (DFA). We discussed the underlying ideas, how to describe them properly and also mentioned some interesting applications and possibilities. Today, we will instead focus on their limitations. More precisely, we will pose a challenge – *recognizing (long) palindromes* – that turns out to be very hard for DFAs. But some, seemingly modest, manipulations do allow us to tackle the palindrome challenge with relative ease. The resulting computational model is called the *Turing machine* and has formed the backbone of theoretical computer science from the 1930s till today.

Agenda:

1. Palindrome challenge
2. How DFAs fail
3. A better approach
4. Turing machines
5. History

3.1 The palindrome challenge

3.1.1 Palindromes

A *palindrome* is a string (word) which backwards reads the same as forwards. Of course, this depends on the alphabet in question. For the (lower-case) latin alphabet, examples are

'anna', 'hannah', 'civic' or 'reliefpfeiler' (German). (3.1)

But palindromes exist for any alphabet. Here is a formal definition.

Definition 3.1 (palindrome). A (finite) string $x = x_0 \cdots x_{n-1}$ over an alphabet Σ is called a *palindrome* if

$$x_{n-1}x_{n-2}\cdots x_1x_0 = x_0x_1\cdots x_{n-2}x_{n-1}.$$

It is easy to verify that the examples from Eq. (3.1) are all palindromes over the lower-case latin alphabet $\Sigma = \{a, b, c, \ldots, z\}$. The following strings are examples of palindromes over the binary alphabet $\Sigma = \{0, 1\}$:

$$11100111, 10111101, \quad \text{and also} \quad 1010101, 1011101.$$

The first two bitstrings are examples of palindromes, where the total number of symbols is even ('anna' and 'hannah' also have this feature). In contrast, the second two bitstrings are palindromes where the total number of symbols is odd ('civic' and 'reliefpfeiler' also have odd length). Looking at these examples suggests that even-length palindromes are slightly more restrictive than odd-length palindromes. Indeed, for odd palindromes, we are allowed to choose the symbol in the very center of the string completely arbitrarily. The following reformulation of Definition 3.1 pinpoints this slight discrepancy.

Lemma 3.2 (palindrome). Fix an alphabet Σ. A (finite) string $x = x_0 \cdots x_{n-1}$ over Σ is called a *palindrome* if and only if

$$x_k = x_{n-1-k} \quad \text{for all } k = 1, \ldots, \lfloor n/2 \rfloor, \tag{3.2}$$

where $\lfloor \cdot \rfloor$ denotes the 'rounding down function', e.g. $\lfloor 1/2 \rfloor = 0$, $\lfloor 1 \rfloor = 1$, $\lfloor 3/2 \rfloor = 1$, etc.

Small mathematical statements, like this one, are called Lemma or 'Hilfssatz' in German. This one is a (very) easy reformulation of Eq. (3.1).

Exercise 3.3 Prove Lemma 3.2.

By looking at Lemma 3.2, we see where the discrepancy between even- and odd-length palindromes comes from. If $|x| = n$ is even, then there are exactly $\lfloor n/2 \rfloor = n/2$ constraints that a palindrome must satisfy. But, for odd-length palindromes, this number of constraints drops down to $\lfloor n/2 \rfloor = (n-1)/2$.

In the remainder of this lecture, we restrict our attention to even-length palindromes. Statements about odd-length palindromes are qualitatively similar, but can differ in one (or more) details. Adapting the ideas discussed here to odd-length palindromes is a good way of understanding what is really going on in this lecture.

But, for now, we conclude this introductory section by exactly counting the number of binary palindromes that have a given length.

Proposition 3.4 Consider the binary alphabet $\Sigma = \{0, 1\}$ and let n be an *even* number. Then,

exponential growth of the number of palindromes

$$\text{Nr. of length-}n\text{ palindromes} = 2^{n/2}. \tag{3.3}$$

This mathematically rigorous statement asserts that the total number of (binary, even-length) palindromes grows *exponentially* in the length n. That is, there are 2 palindromes of length $n = 2$ (00, 11), 4 palindromes of length $n = 4$ (0000, 0110, 1001, 1111), and, for instance, 1024 palindromes of length $n = 20$ (we won't list them all here). The main gist is that Eq. (3.3) grows

extremely quickly with n. Mathematical statements, like this one, are called 'Proposition', because they are more interesting than a 'Lemma', but still don't quite deserve the title 'Theorem'.

Proof of Proposition 3.4. We start with the reformulation of palindrome properties from Lemma 3.2. By assumption, the length n is even, so each palindrome must obey exactly $n/2$ constraints (3.2). We can rewrite them as

$$\begin{aligned} x_{n/2+1} &= x_{n/2}, \\ x_{n/2+2} &= x_{n/2-1}, \\ &\vdots \\ x_{n-1} &= x_0. \end{aligned}$$

These equalities highlight that the choice of the first $n/2$ bits $(x_0, \ldots, x_{n/2-1})$ completely determines what the final $n/2$ bits must look like. Other than that, there are no restrictions. We have, in fact, established a one-to-one relation between palindromes of length n and bitstrings of length $n/2$ (Every length-$n/2$ bitrstring can be completed to form a length-n palindrome and, conversely, every length-n palindrome can be chopped up into two pieces. And, knowing the first piece unambiguously characterizes the second one). But, it is easy to count the number of (distinct) length-$n/2$ bitstrings. There are two possible choices (0 or 1) for each bit which accumulates to a total of

$$\underbrace{2 \times 2 \cdots 2 \times 2}_{n/2 \text{ times}} = 2^{n/2}$$

distinct choices. ∎

This proof is our first example of a *counting argument*. Sometimes, though not always, disputes in theoretical computer science can be settled by simply counting the number of possibilities. Working largely with 0s and 1s turns out to be a major blessing in this regard (by contrast, it is actually impossible to count the number of real-valued numbers between 0 and 1). For the task at hand, the counting can readily be generalized to cover odd-length palindromes and/or more general alphabets.

counting argument

Exercise 3.5 (number of more general palindromes).

1. Generalize Proposition 3.4 (and the underlying proof) to *odd-length* palindromes.
2. Generalize Proposition 3.4 (and the underlying proof) to palindromes over an alphabet Σ comprised of N symbols.
3. Combine 1., 2. and Proposition 3.4 into a single growth formula that is valid for *all* (finite) alphabet sizes $|\Sigma| = N$ and both, even and odd, string length n.

3.2 Attempting to identify palindromes with finite state automata

We will now show that DFAs are very bad at recognizing palindromes. Recall the concept of a *(deterministic) finite state automaton* (DFA) from the last lecture. These are simple computing devices whose inner working is governed by a finite set of internal states Q, as well as a (deterministic) transition function $\delta : Q \times \Sigma \to Q$ between these states. The transition function reacts to (external) symbols in a given alphabet Σ. This allows us to process strings $x = x_0 \cdots x_{n-1}$ over Σ by initializing the DFA in a specified starting state (called q_0) and iteratively applying state transitions according to the symbols $x_k \in \Sigma$ ($0 \leq k \leq n-1$) that make up the string. If we happen to end up at a pre-determined accept state $q \in F \subseteq Q$, we say that our DFA accepts string x. Formally, a DFA is described by a 5-tuple $M = (Q, \Sigma, \delta, q_0, F)$, where Q is the set of states and $F \subseteq Q$ is a subset of accepting states.

DFAs can, and should, be used to identify interesting structure within strings of symbols. Last week, for instance, we constructed a DFA that accepts bitstrings with *odd parity*, i.e. $x = x_0 \cdots x_{n-1}$ with $x_k \in \{0,1\}$ and $x_{n-1} = 1$, regardless of the actual length n. Another example is bitstrings that contain an odd number of 1s (parity of sums), which we left as an exercise last time. And, we will see several more examples when we talk about regular languages next week. For now, it suffices to appreciate that it is often possible to develop DFAs that accept (bit-)strings with certain structural properties. But, does this also work for identifying (even) palindromes?

The perhaps surprising answer is *no, not really!* Here, we content ourselves with illustrating what can go wrong when we try to design DFAs that accept palindromes. For now, it is enough to consider bitstrings of (even) length n, where n is a somewhat large number, e.g. $n = 550$. The task is to design a DFA that accepts length-n bitstrings if and only if they are palindromes. But this is already a challenging endeavor, because Proposition 3.4 tells us that there are exponentially many length-n palindromes. And, to make matters worse, they are apparently structureless. As shown in the proof of Proposition 3.4, every length-$n/2$ bitstring can be completed to form a palindrome that is twice as long. And, there is one, and only one, correct way to do that. Reverse the ordering and append it to the original string:

$$\underbrace{x_0 \cdots x_{n/2-1} \in \{0,1\}^{n/2}}_{\text{bitstrings of length } n/2} \mapsto \underbrace{x_0 \cdots x_{n/2-1} x_{n/2-1} \cdots x_0 \in \{0,1\}^{n}}_{\text{palindromes of length } n}.$$

The problem is, that the palindrome structure only manifests itself after we have processed exactly one half of the input bitstring in question. And, to make matters worse, (even) palindromes are highly 'case-sensitive' as well. This has a very unfortunate consequence. In order to check for palindromes, our DFA is forced to 'remember' every possible configuration of $n/2$ bits in order to correctly check consistency on the final $n/2$ bits. And, the number of different length-$n/2$ bitstrings scales as $2^{n/2}$, see Proposition 3.4. In order to remember all these configurations, our hypothetical DFA requires at least $2^{n/2}$

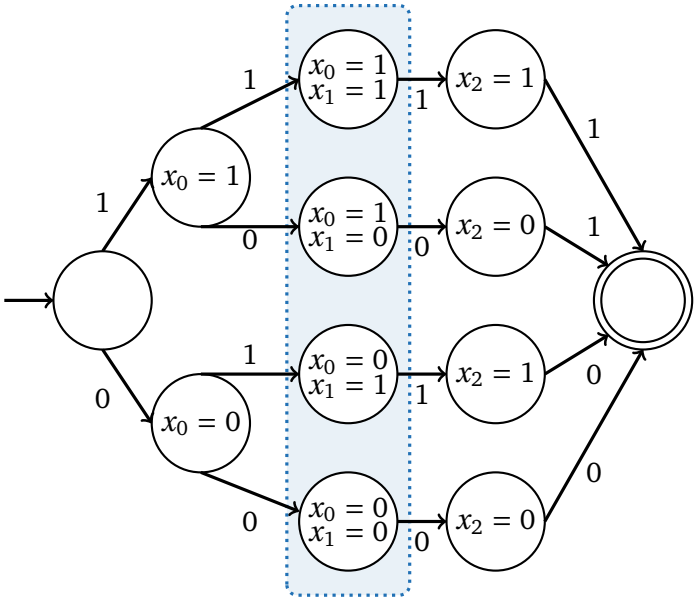


Figure 3.1 *Illustration of a DFA that accepts palindromes of length $n = 4$:* transition arrows not shown here would lead to a rejection of the palindrome property. This could be implemented, for instance, by self loops in the second and third layer.

distinct internal states (in fact, it will require many more than that). Figure 3.1 illustrates such a construction for $n = 4$. In other words: the number of internal states must scale (at least) *exponentially* in half the length of the bitstrings.

exponential growth of DFA states

This is scary growth behavior! E.g. for $n = 550$, this lower bound on the number of distinct states becomes $2^{n/2} = 2^{225} = 5.4 \times 10^{67}$ – which is twice as large as the total number of atoms in the milky way galaxy. This seems to indicate that palindrome recognition with DFAs quickly becomes *very* expensive. Not in terms of runtime (the number of steps), which is always n, but in terms of internal hard-wired functionalities (the number of states required). We emphasize that the problem of designing DFAs that accept palindromes of a fixed (and known) length n is strictly less challenging than developing a DFA that accepts palindromes of all possible lengths. And we just discovered that this restricted problem already looks very challenging. Chances are that it doesn't get any simpler if we allow for varying input length n as well.

However, care must be taken before jumping to conclusions.

Warning 3.6 Failing to construct a reasonable DFA does not (yet) imply that such DFAs cannot exist. ▪

But for palindromes, and other similar-looking structures, it is actually possible to rigorously prove that any DFA – no matter how ingenious the design – must fail. We will present a mathematically rigorous proof in a future lecture.

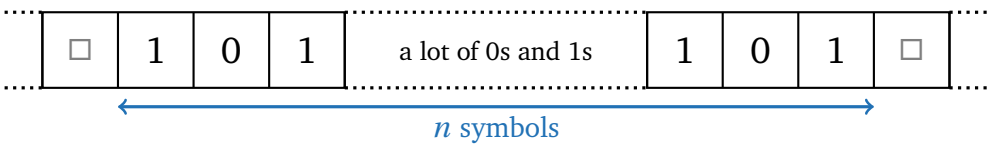


Figure 3.2 *Setup for identifying (even) palindromes:* We are given a bitstring x of even length n and must check whether it is a palindrome or not. We envision that the bitstring in question is written onto a very long, checkered strip of paper. The square symbol (□) denotes empty paper space at either side of the input.

3.3 A better approach to identify (even) palindromes

We have just seen that DFAs struggle at identifying palindromes of largish length n. The reason for that is that they suffer from two crucial limitations:

1. They must process input symbols sequentially from left to right in one go (think: no random input access)
2. They can only read input symbols, but don't have a way to write down intermediate results (think: no actual memory).

We now discuss a simple procedure that can identify (even) palindromes by using a simple computational model that is allowed to do both of these things. For illustrative purposes, we assume that the length-n bitstring in question is written down somewhere in the middle of a very long, checkered strip of paper. In particular, we assume that the strip of paper contains much more than n checkered boxes. This means that there is empty paper space towards the left and right of our length-n bitstring. See Figure 3.2 for an illustration. We denote empty paper space by the *square symbol* '□' which happens to look a bit like an empty box. The palindrome string visualized in Figure 3.2 contains 0, 1 and □:

□ denotes empty space

$$\square \cdots \square 101 \cdots\cdots 101 \square \cdots \square.$$

We also assume that we can only process boxes one at a time. Processing may involve reading a symbol and replacing it with a possibly different symbol. And, after we have done that, we can only move one box to the left, or one box to the right. Suppose that we start reading and processing the input string somewhere in the middle, where all the relevant information is stored. How could we check if bitstring x is an (even) palindrome? Well, we could first traverse box-by-box to the left until we encounter the first □-symbol. This tells us that we have found the beginning of the actual bitstring. Moving to the right by one square allows us to read the first bit:

$$\square \cdots \square \underline{1}01 \cdots\cdots 101 \square \cdots \square.$$

Here, the blue underline indicates our current focus (box) of attention. In our example, the first bit is a 1. And so, if x is indeed a palindrome, the final bit

better be a 1 as well ($x_{n-1} = x_0 = 1$). But, this equivalence between two bits is easy and cheap to check. We 'remember' that the first bit is 1, then move to the very end of the bitstring to check whether $x_{n-1} = 1$. If this is not the case, x cannot be a palindrome and we are done. Else if $x_{n-1} = 1$, the first palindrome constraint from Lemma 3.2 is satisfied and we can move on to check the next one ($x_1 = x_{n-2}$). But note that this constraint, and all other remaining ones, do neither include x_0, nor x_{n-1} anymore. So, we may as well erase the first and last bit to make the remaining bitstring smaller and save resources. This can be achieved by replacing the first bit by □ ('erasure') just after reading and remembering it. Likewise, we also replace the last bit by □ ('erasure') immediately after we compare its value to the bit we remembered. For the example at hand, such a subroutine looks like

$$\begin{array}{ll} \square\cdots\square\underline{1}01\cdots\cdots101\square\cdots\square & \text{(find the first bit),} \\ \square\cdots\square\square\underline{0}1\cdots\cdots101\square\cdots\square & \text{(remember first bit, erase } x_0\text{, move right),} \\ \square\cdots\square\square01\cdots\cdots10\underline{1}\square\cdots\square & \text{(find the last bit),} \\ \square\cdots\square\square01\cdots\cdots1\underline{0}\square\square\cdots\square & \text{(check equivalence, erase and move left).} \end{array} \tag{3.4}$$

Here, the first and last bit are both 1 and we don't find a palindrome violation immediately. But we do end up with a bitstring of length $n-2$, because we have erased the first and last bit. And it is easy to check that the original string x is an (even) palindrome if and only if this slightly shorter string is. But, we already know how to make progress on this slightly simpler question: simply apply the subroutine from Eq. (3.4) all over again. This will either find a violation of the second palindrome condition ($x_1 = x_{n-2}$), or produce an even shorter string of length $n-4$. All in all, there might be up to $n/2$ iterative applications of this one-sided test and the bitstring x is an (even) palindrome if and only if it passes all of them. So, to summarize, we have found a simple procedure that iteratively compares pairs of bits to check for palindromes. This procedure avoids the issues we faced for DFAs by sweeping back and forth across the input and erasing input symbols that have already been processed. This not only makes each subroutine invocation cheaper than the previous one, but also allows us to find the next relevant pair of input bits with relative ease. In fact, our procedure is so simple that it can be executed with a *constant* number of internal states – 10 to be exact (see Figure 3.4 below for details) – regardless of the actual length n of the bitstrings in question. As shown in detail below, we use these internal states to 'remember whether we have seen a 0 or 1' and similar simple things. The constant number of internal states is astronomically better than the $2^{n/2}$ internal states that had been required for our DFA attempt.

iterative checking procedure

constant number of internal states

It is also interesting to estimate the maximum number of steps that might be required to check (even) palindrome for length-n bitstrings. We call this the *worst-case runtime*. Suppose that we are facing a bitstring of length $k \in \{2, 4, 6, \ldots, n\}$, sandwiched by □-symbols. Then, identifying the leading bit can take up to $k+1$ iterative steps (if we start at the very last bit symbol, it can take k steps to find the first □-symbol towards the left of the string and

one additional step to move back one square). This bounds the cost for the first step (sweep left) in Eq. (3.4). The second operation only requires a single computation step, while the third one (sweep right) requires k steps ($k-1$ to find the first $\square$-symbol towards the right and one additional step to move back one square). The fourth and last operation is again cheap and always requires a single step. All in all, we require

$$N(k) = (k+1) + 1 + k + 1 = 2k + 3 \quad \text{for} \quad k \in \{0, 2, 4, 6, \ldots, n\}$$

elementary steps to execute the subroutine displayed in Eq. (3.4) on a length-k string and either find a palindrome violation or end up with a string of length $k-2$. Since we start with a length-n string, the total number of steps is at most

$$\begin{aligned} N_{\text{tot}} =& N(n) + N(n-2) + N(n-4) + \cdots + N(2) \\ =& \sum_{j=1}^{n/2} N(2j) = \sum_{j=1}^{n/2} (2 \times (2j) + 3) \\ =& 4 \sum_{j=1}^{n/2} j + 3 \sum_{j=1}^{n/2} 1 \\ =& 4 \frac{n}{4} \left(\frac{n}{2} + 1 \right) + 3 \frac{n}{2} \\ =& \frac{1}{2} n^2 + \frac{5}{2} n. \end{aligned}$$

Note that the actual runtime can be much lower if we happen to find a palindrome violation early on. In this derivation, we have used $\sum_{j=1}^{n/2} j = (n/2)(n/2+1)/2$ (Gauss summation) and $\sum_{j=1}^{n/2} 1 = n/2$ (adding a total of $n/2$ ones produces $n/2$) to obtain a nice and exact closed-form expression for the sums involved.

Exercise 3.7 (Gauss summation). Show that $\sum_{j=0}^{M} j = M(M+1)/2$ is true for any natural number $M \in \mathbb{N}$.

The worst-case number of steps $N_{\text{tot}} = n^2/2 + 5n/2$ scales *quadratically* in the length n of bitstrings to be processed. This is worse than the linear runtime ($N_{\text{tot}} = n$) a DFA would achieve. But, on the plus side, we do now get by with a fixed and constant number of hardwired internal states.

quadratic (worst-case) runtime

Exercise 3.8 (Identifying odd palindromes). How would you have to modify our procedure to be able to check odd-length palindromes?

3.4 Turing machines

3.4.1 Intuitive definition

The computing model we designed in the previous section contains three components:

finite state control, tape and tape head

1 The *finite state control:* at each instant, this component is in one of a finite number of states. (E.g. remember that the first bit was 1 *and* move towards the right)

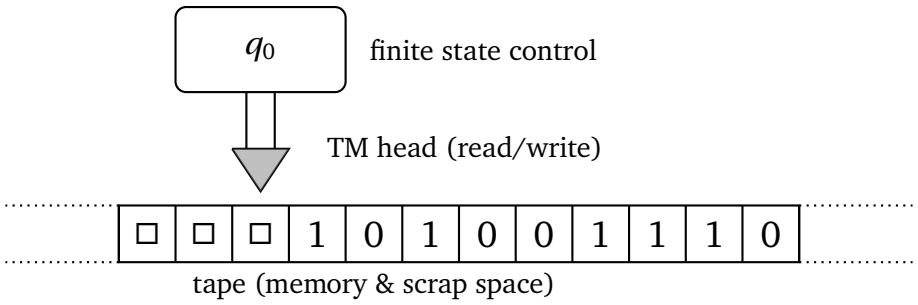


Figure 3.3 *Schematic illustration of a TM.*

2 The *tape:* this component consists of an infinite number of *tape squares*. Each of these squares can store a single symbol taken from a specified alphabet Σ, but can also be empty ($\square$). Furthermore we assume that the tape is infinite both to the left and to the right.
3 The *tape head:* the tape head keeps track of the current position. It can move either left (L) or right (R) on the tape. At the start of each computational step, the tape head accesses exactly one tape square. It can read which symbol is stored there and can also write a new symbol into that square.

See Figure 3.3 for a visualization. This model of computation is called the *Turing machine model*. Today, the terminology and analogies may seem a bit outdated (it is reminiscent of tape recorders and the like), but the underlying idea very much isn't. We will discuss important connections to machine language and algorithms in Sec. 3.4.4 below. For now, we emphasize that the action of a Turing machine at each instant is determined by the state of the finite state control together with the single symbol it has currently access to ('computation is local'). Thus, the action of the machine is determined by a finite number of possible transitions: one for each state/symbol pair. This is very reminiscent of a DFA. And similar to DFAs, finite resources are enough to completely specify a Turing machine. And yet, we have just seen that the resulting computing power can be much more impressive than what a DFA could ever hope to do. Again similar to DFAs, Turing machines can perform computations on input strings with varying length n. But they may loop back and forth a lot which can lead to runtimes that are much larger than n (e.g. our palindrome checking machine had a quadratic worst-case runtime). But in order to even talk about runtime, we must provide Turing machines with the option to stop its computation and actually produce an outcome. We ensure this by including two special types of internal states: *accept states* and *reject states*. If the machine enters one of these states, it stops immediately and either *accepts* or *rejects* the input string. We also call such states *halting states*.

Turing machine model

accept and reject states

It is here, where we start seeing notable deviations from DFA computation. For DFAs, it was enough to only specify accept states. This is, because the number of DFA computational steps is always equal to the input length n.

This guaranteed stopping time has allowed us to implicitly define reject states as those states which are not accept states. For Turing machines, this is not possible anymore, because the actual number of computing steps is not stringently pre-defined. In fact, even with accept and reject states, it is entirely possible that certain Turing machine computations do not terminate at all. This issue will be the content of a future lecture.

3.4.2 Formal definition

We are now ready to convert the intuitive notion of a Turing machine into a mathematically rigorous definition.

Definition 3.9 (Turing machine (TM)). A *Turing machine* (TM) is a machine that can either accept or reject strings by locally processing (read/write) symbols stored on an infinite tape. It is fully characterized by

- a finite (nonempty) set of internal *states* Q;
- the alphabet Σ of symbols it can process, called the *input alphabet*;
- another alphabet Γ, called the *tape alphabet*, that also contains the blank symbol '$\square$': $\Sigma \cup \{\square\} \subseteq \Gamma$;
- a *transition function* $\delta : (Q \setminus \{q_{\text{accept}}, q_{\text{reject}}\}) \times \Gamma \to Q \times \Gamma \times \{L, R\}$;
- a designated start state $q_0 \in Q$;
- a (single) designated accept state $q_{\text{accept}} \in Q$; and
- a (single) designated reject state $q_{\text{reject}} \in Q$.

Formally, we identify a TM with the *7-tuple* $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$. TMs are 7-tuples

We know several of these objects already from our formal definition of DFAs. The set of states Q, the input alphabet Σ, the start state q_0 and the accept state q_{accept} are all virtually identical (the only difference is that DFAs allow for multiple accept states, while we only allow a single one for TMs). Other objects, like the tape alphabet Γ (think: input alphabet plus blank symbol) and the single reject state q_{reject}, are straightforward extensions of these DFA concepts.

It is mainly the transition function δ that becomes more involved. In fact, there is quite a lot to unpack here. First, note that entering either the accept and the reject state immediately terminates the computation. So, we can exclude these two states as possible inputs for the transition function (a transition away from q_{accept} or q_{reject} is never going to happen). This is the meaning of $Q \setminus \{q_{\text{accept}}, q_{\text{reject}}\}$ which denotes the set of all states except q_{accept} and q_{reject}. Next, we should make sure that the TM is capable of processing, reading and writing blank tape symbols $\square$, as well as symbols from the input alphabet Σ. This is why we allow the second input of δ to be selected from the tape alphabet Γ which is guaranteed to contain all these symbols. TM transition function

But, it is the output of the transition function, where things truly get interesting. Similar to DFAs, a TM can change its internal state to another element of Q (e.g. 'remembering' that we have seen a 1 before), but they can also replace the current tape symbol with a new symbol taken from the tape alphabet (e.g. 'erasing' a 1 by replacing it with $\square$) and it can move its tape head towards

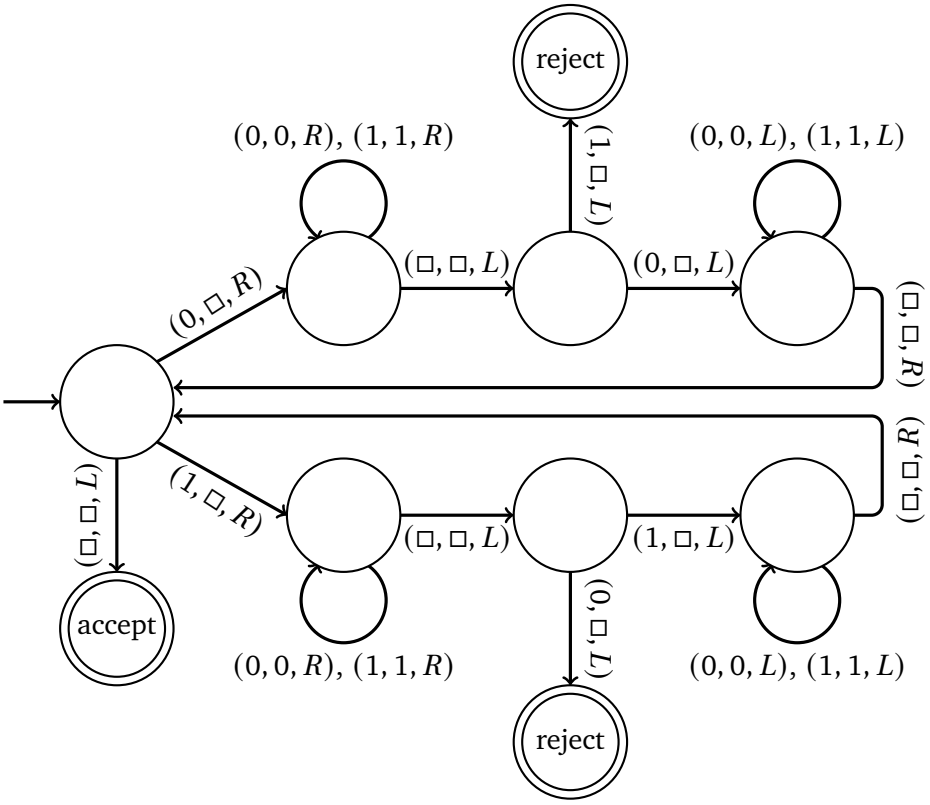
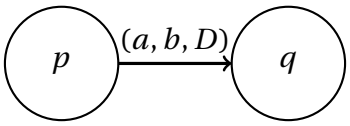


Figure 3.4 *State diagram for the palindrome checking Turing machine (even length) from Section 3.3.* We assume that we already start at the location of the very first bit. If it is 0, we enter into the upstairs trajectory. If it is 1, we enter the downstairs trajectory instead. Either trajectory can only be escaped via the central arrows if the last bit happens to be equal to the first one (and we erase it). Otherwise, we enter one of two reject states (we also could subsume these in a single circle). Transitions not explicitly depicted are not supposed to happen if the input has even length and is properly formatted.

the left (L) or towards the right (R). A TM transition function must specify these three outputs for every possible state/input pair. We can summarize all these things by writing $\delta_{\text{TM}} : (Q \setminus \{q_{\text{accept}}, q_{\text{reject}}\}) \times \Gamma \to Q \times \Gamma \times \{L, R\}$.

This transition function looks much more complicated than its DFA counterpart ($\delta_{\text{DFA}} : Q \times \Sigma \to Q$), because, well, it is. However, we can still visually express the transition function by means of *state diagrams*. Internal TM states can still be represented by circles and special states – like q_0, q_{accept} and q_{reject} – can be highlighted as usual. It is the transition arrows that must become more expressive. Suppose that our transition function satisfies $\delta(p, a) = (q, b, D)$, where $p, q \in Q$ are states, $a, b \in \Gamma$ are tape alphabet symbols and $D \subset \{L, R\}$ tells us if we should move location towards the left ($D = L$) or one location towards the right ($D = R$). We can represent this property visually by writing

TM state diagrams



Each transition arrow is labeled by a 3-tuple in $\Gamma \times \Gamma \times \{L, R\}$. The first entry tells us which tape symbol a must be read to trigger the transition in question. The second entry singles out the tape symbol b we replace a with. And the third entry $D \in \{L, R\}$ tells us whether we move the tape head to the left or to the right. Figure 3.4 visualizes a transition diagram for our palindrome checking TM. (To keep things as simple as possible, we have only written down transition arrows that ought to appear when the TM is working as intended. A wrongly formatted input, for instance, may 'break' this TM.)

3.4.3 Turing machine computations

A TM is fully characterized by the 7-tuple $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$ introduced in Definition 3.9. The machine is intended to process input strings x of arbitrary length n. These input symbols are written somewhere in the middle of the tape. Before, we also suggested that the tape head is initially focused on some box in the middle. But this is not very precise. From now on, we assume that the TM starts with its head immediately left from the first input symbol and the initial state is q_0 (starting state). A possible initial configuration has been visualized in Figure 3.3 above. For the sake of completeness, we also point out that we can handle 'empty inputs'. An empty string ε would simply translate to only squares on the tape.

initial configuration

TM initialization

After proper initialization, the TM is ready to start working. And it does so in *computing steps*. Each step is completely determined by the content of the tape and the transition function. In other words: the TMs we consider here are *deterministic* models of computation. It involves reading a symbol with the TM head (read), replacing it with another symbol (write), changing the internal state of the TM and moving one square to the left or right. This step-wise computation continues as long as the TM does not enter one of the two designated halting states $q_{\text{accept}}, q_{\text{reject}}$:

computing steps

TM termination

1 If the TM enters the state q_{accept}, it halts (i.e. it stops computing) and *accepts* the input string x.
2 If the TM enters the state q_{reject}, it halts (i.e. it stops computing) and *rejects* the input string x.

Accepting and rejecting are two mutually exclusive possibilities, but there is a third one:

3 The TM never reaches a halting state and runs forever on input string x.

3.4.4 Specifications

This extension of DFA state diagrams allows us to represent any TM computation visually. Figure 3.4, for instance, captures the inner working of our palindrome checking TM visually. This is actually a state diagram example of a very simple TM. The TMs we will be most interested in are going to be much more complicated.

Instead, this is the right time for an additional level of abstraction. The usual way to describe TMs is in terms of pseudo-code or high-level descriptions, like the one we actually use to devise our palindrome checking procedure. The broad summary term for such descriptions is *algorithms*. We don't describe TMs by completely specifying transition functions, but by providing an intuitive guide on how the 'TM programs' we envision should be executed.

description in terms of algorithms

A useful analogy is as follows: TMs are capable of executing basic instructions like read a symbol, write a symbol, move left on a register, move right on a register. With state transitions, we can also implement simple 'if, then'-conditions. These sets of instructions are not dissimilar from a *machine language*. This is a sequence of simple instructions to read from memory into one of a finite number of registers, write a register's content to memory, perform basic arithmetic operations (e.g. adding two registers) and control instructions that perform conditional actions. All these operations can also be simulated by a TM. Showing this rigorously does take some work, but intuitively this should not be too surprising. It does, however, have profound consequences: *it is possible to simulate programs in your favorite programming language using a Turing machine!*

connection to machine language

In summary, one can think of the TM as a simplified modern computer. The tape corresponds to a computer's memory, while finite state control and transition function correspond to the computer's central processing unit (CPU). However, it is best to think of TMs as a formal way to describe algorithms. We will see that it can be very useful to express algorithms in terms of TMs, because it allows us to reason about them mathematically. (This is similar to expressing an algorithm in an actual programming language in order to execute it on an actual computer).

3.5 History

The Turing Machine is named after *Alan Turing* (1912 – 1954), an English mathematician and computer scientist. His life is a fascinating, but ultimately also tragic story. Today, Alan Turing is widely considered to be a father of both computer science and artificial intelligence. Among many other things, he wrote a chess program for computers that didn't yet exist (1948) and devised the *Turing test*, originally known as the imitation game (1950). During World War II, he worked at Bletchley Park (the codebreaking center of the United Kingdom) and managed to reliably crack the Enigma – a German cipher device used to coordinate U-boat attacks in the Atlantic. The success of this top secret project ultimately tipped the Naval scales in favor of the allies[1]. To this date, the highest distinction for any computer scientist is the *ACM A. M. Turing Award*, widely regarded as the 'Nobel Prize of Computer Science'.

Alan Turing

Turing Award

The hypothetical machines we discussed today were introduced by Turing in a math paper from 1936. His original motivation was the resolution of an important mathematical problem at the time, the so-called 'Entscheidungsproblem'. We will talk more about these aspects and implications in a future lecture. The language and analogies used to illustrate Turing Machines still convey an aura from the first half of the 20th century. But, in fact, it was human computers that inspired him – a very real and important job, often occupied by females, before digital computers became commercially available. In Turing's own words:

> *"The idea behind digital computers may be explained by saying that these machines are intended to carry out any operations which could be done by a human computer. The human computer is supposed to be following fixed rules; he has no authority to deviate from them in any detail. We may suppose that these rules are supplied in a book, which is altered whenever he is put on to a new job. He has also an unlimited supply of paper on which he does his calculations."*
> Alan Turing, 1950

Problems

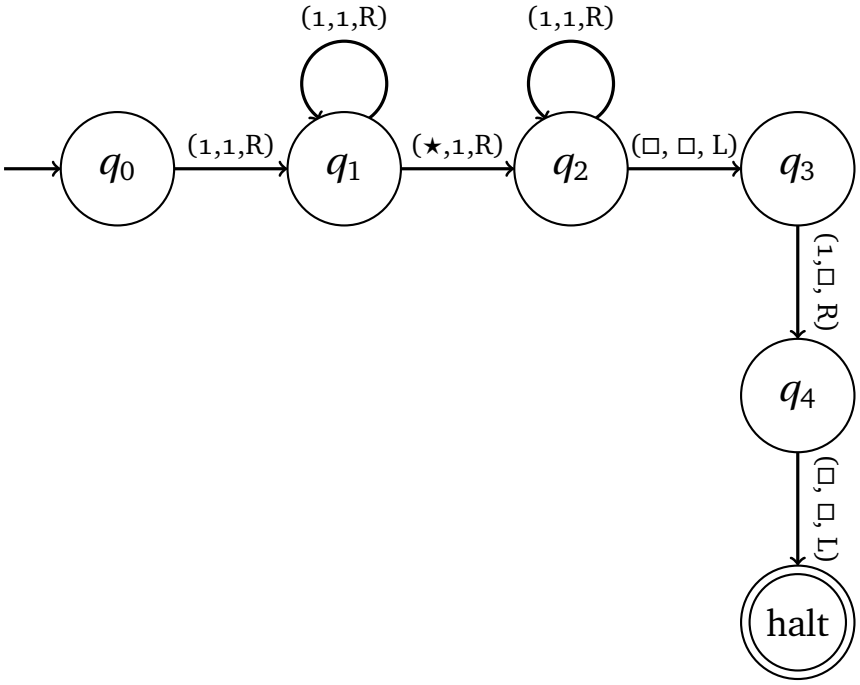
Problem 3.10 (state diagram to Turing machine I). We consider Turing machines over the alphabet $\Sigma = \{1, \square, *, \star\}$, where (i) 1-symbols are used to write inputs & outputs, $\square$ denotes 'empty' tape space, $*$ is used to cross out input symbols that have already been processed and $\star$ denotes a *certain* arithmetic operation – the goal of this exercise, and the next one, is to find out which arithmetic operation it is.

Throughout this exercise (and the next one), we use *unary encoding* for inputs and outputs. That is, numbers $n \in \mathbb{N}$ are represented by strings of exactly n ones: $1 \leftrightarrow 1$, $2 \leftrightarrow 11$, $3 \leftrightarrow 111$, $4 \leftrightarrow 1111$ and, more generally,

[1] The movie *The Imitation Game* (2014) captures this episode of Turing's life in motion picture.

$n \leftrightarrow 1^n = \underbrace{1 \cdots 1}_{n \text{ times}}.$

The Turing machine in question is characterized by the following state diagram:



The arrows are labelled by 3-tuples that indicate 'read', 'write' and 'move' (L for left, R for right and S for stop). E.g. (★,1,R) means 'read ★', 'write 1' and 'move right'. In the following, 'underline' denotes the initial and final positions of the TM head. Which of the following statements is correct?

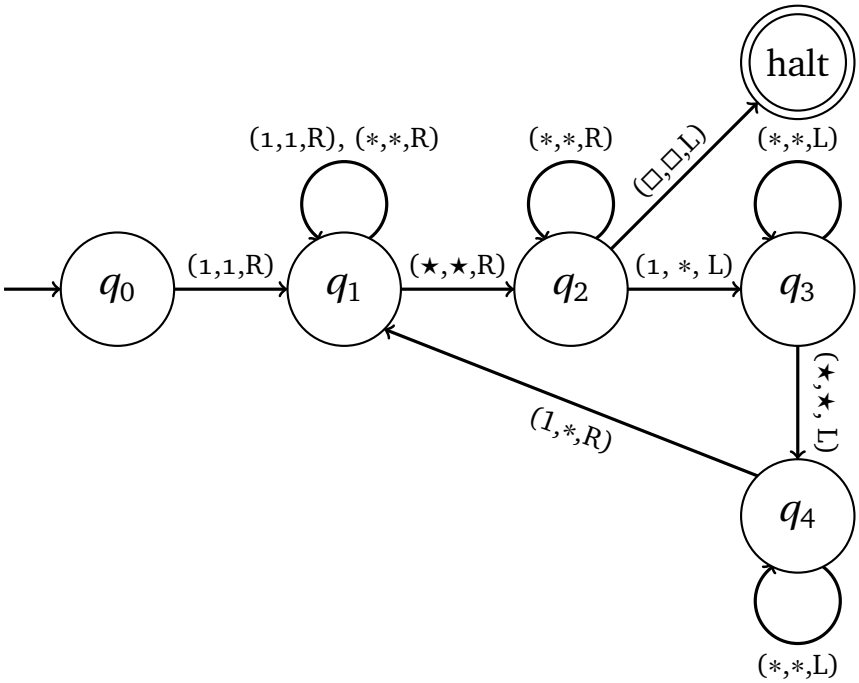
- **a)** (★ = +) the TM computes the *sum* of two numbers (in unary encoding) e.g. input $\underline{1}$ 1 1 ★ 1 1 □ produces output 1 1 1 1 1 $\underline{□}$ □.
- **b)** (★ = −) the TM computes the *difference* of two numbers (in unary encoding), e.g. input $\underline{1}$ 1 1 ★ 1 1 □ produces output 1 ∗ ∗ ★ ∗ $\underline{∗}$ □.
- **c)** (★ = ×) the TM computes the *product* of two numbers (in unary encoding), e.g. input $\underline{1}$ 1 1 ★ 1 1 □ produces output 1 1 1 1 1 1 $\underline{□}$.
- **d)** (★ = /) the TM computes the *division* of two numbers (in unary encoding), e.g. input $\underline{1}$ 1 1 1 1 1 ★ 1 1 □ produces output 1 1 1 ∗ ∗ ∗ ★ ∗ ∗ $\underline{□}$.

Problem 3.11 (state diagram to Turing machine II). We consider Turing machines over the alphabet $\Sigma = \{1, □, ∗, ★\}$, where (i) 1-symbols are used to write inputs & outputs, □ denotes 'empty' tape space, ∗ is used to cross out input symbols that have already been processed and ★ denotes a *certain* arithmetic operation – the goal of this exercise, and the previous one, is to find out which arithmetic operation it is.

Throughout this exercise (and the previous one), we use *unary encoding* for inputs and outputs. That is, numbers $n \in \mathbb{N}$ are represented by strings of exactly n ones: $1 \leftrightarrow 1$, $2 \leftrightarrow 11$, $3 \leftrightarrow 111$, $4 \leftrightarrow 1111$ and, more generally,

$n \leftrightarrow 1^n = \underbrace{1 \cdots 1}_{n \text{ times}}.$

The Turing machine in question is characterized by the following state diagram:



The arrows are labelled by 3-tuples that indicate 'read', 'write' and 'move' (L for left, R for right and S for stop). E.g. (★,1,R) means 'read ★', 'write 1' and 'move right'. In the following, 'underline' denotes the initial and final positions of the TM head. Which of the following statements is correct?

- **a)** (★ = +) the TM computes the *sum* of two numbers (in unary encoding) e.g. input $\underline{1}$ 1 1 ★ 1 1 □ produces output 1 1 1 1 1 $\underline{□}$ □.
- **b)** (★ = −) the TM computes the *difference* of two numbers (in unary encoding), e.g. input $\underline{1}$ 1 1 ★ 1 1 □ produces output 1 ∗ ∗ ★ ∗ $\underline{∗}$ □.
- **c)** (★ = ×) the TM computes the *product* of two numbers (in unary encoding), e.g. input $\underline{1}$ 1 1 ★ 1 1 □ produces output 1 1 1 1 1 1 $\underline{□}$.
- **d)** (★ = /) the TM computes the *division* of two numbers (in unary encoding), e.g. input $\underline{1}$ 1 1 1 1 1 ★ 1 1 □ produces output 1 1 1 ∗ ∗ ∗ ★ ∗ ∗ $\underline{□}$.

4. Decision problems and languages

Date: October 24, 2024

Agenda

Agenda:

1. 3 points of view on computational problems
2. Regular languages
3. (Semi-)decidable languages
4. Church-Turing thesis

In the last two lectures, we have seen two models of computation: *finite state automatas* (DFAs) and *Turing machines* (TMs). With these computational models at the ready, we can begin a deep dive into the heart of *computability* theory. Today, we are interested in the ultimate limits for (any) computation.

We first introduce general decision problems (yes/no questions) and two ways on how to reformulate them. This will motivate the definition of a *language* which simply encompasses a set of strings. Then, we will explore what types of decision problems (languages) can be computed by a DFA and, perhaps more interestingly, what kind of problems really cannot be computed by a DFA. We will then move on to discuss computational problems that can be solved with TMs. Perhaps surprisingly, we will see that there are also limitations. And shockingly, there might be no conceivable way to overcome them.

4.1 Three points of view on computational challenges

4.1.1 Decision problems

In science and engineering, we ultimately want to get computing architectures to answer interesting, challenging, or perhaps simply tedious, questions. One of the most basic types of questions is a *yes/no question*, aka accept/reject. Problems of this type are also called *decision problems*, because a single yes/no decision is all it takes. We have already seen several examples of decision problems. Here is an example that already featured prominently in Lecture 3, but today we formulate it in a slightly different fashion. Define the *reverse*

decision problem

operation x^{R} recursively by setting $\varepsilon^{\mathrm{R}} = \varepsilon$ (the empty string is its own reverse) and $(aw)^{\mathrm{R}} = w^{\mathrm{R}}a$ for every $w \in \Sigma^{\times n}$ ($n \in \mathbb{N}$) and $a \in \Sigma$. In words: upon reversing, every symbol we add on the left must move to the very right.

reverse operation R

Example 4.1 (binary palindrome – decision problem). Given a bitstring x, decide whether $x^{\mathrm{R}} = x$. ▪

4.1.2 Computing a Boolean function

Equivalently, we can also encode decision problems into a logical function. This function, let's call it f, takes a problem instance as input and outputs exactly one of two truth values:

- 1: which stands for true, accept, yes, etc. or
- 0: which stands for false, reject, no, etc.

Functions $f(x)$ with a single bit (truth value) output are called *Boolean functions*. And being able to consistently solve a decision problem is equivalent to consistently being able to compute a Boolean function.

Example 4.2 (binary palindrome – Boolean function). The decision problem from Example 4.1 is equivalent to computing the following Boolean function on bitstrings:

$$f(x) = \begin{cases} 1 & \text{if } x^{\mathrm{R}} = x, \\ 0 & \text{else.} \end{cases}$$

▪

4.1.3 Languages

There is yet another point of view on decision problems and Boolean function computations, respectively. Conceptually, this approach is closer aligned with the basic notational concepts that underpin theoretical computer science. For the scope of this course, the most elementary building blocks of information are symbols taken from an *alphabet* Σ. These symbols can be subsequently combined to form *strings* – think of words – which are (finite) sequences of symbols $x = x_0 \cdots x_{n-1}$ with $x_k \in \Sigma$. If we keep on following this route, it seems natural to associate a *language* with a collection of strings (words).

Definition 4.3 (language). A *language* over alphabet Σ is a collection (set) of strings over Σ.

languages

Two very simple, yet useful, languages are the *empty language* $\emptyset$ (which doesn't contain a single word), as well as the *language of all strings*, which we denote by Σ^*. For our favorite alphabet $\Sigma = \{0, 1\}$, this definition simplifies to

$$\{0,1\}^* = \{\varepsilon, 0, 1, 00, 01, 10, 11, 000, 001, \ldots\}. \tag{4.1}$$

Note that this language is actually infinitely large (but in a countable fashion). All conceivable binary languages are subsets of this complete set of strings.

Here are two more interesting examples of binary languages:

$$A = \{0000, 0011, 1100, 1111\} \subset \{0,1\}^*,$$
$$\textsc{Parity} = \left\{x \in \{0,1\}^* : \text{ the last bit of } x \text{ is a } 1\right\} \subset \{0,1\}^*.$$

Both are strict subsets of the set of all binary strings. the first language is finite, while the second one is not. We are now ready for presenting the equivalence between Boolean functions and languages. Let $f : \Sigma^* \to \{0,1\}$ be a Boolean function that encodes some decision problem of interest. We can then define the language

$$L_f = \{x \in \Sigma^* : \ f(x) = 1\} \subseteq \Sigma^* \tag{4.2}$$

to be the subset of all strings for which f evaluates to 1 (true). Given this definition, it is easy to see that decision problems, Boolean functions and languages are three ways to describe the same underlying concept. In fact, we can use them interchangeably. Different formulations highlight different aspects of the same computational problem.

decision problems and languages are equivalent

Example 4.4 (binary palindrome – language). Computing the Boolean function from Example 4.2 is equivalent to deciding membership in the language

$$\textsc{Palindrome} = \left\{x \in \{0,1\}^* : \ x^{\mathrm{R}} = x\right\} \subset \{0,1\}^*.$$

▪

For the remainder of this lecture, we will mainly talk about languages. We will see that certain computing platforms can only decide a small fraction of all conceivable languages. This is equivalent to saying that certain computing platforms can only answer a small fraction of all possible yes/no questions.

4.2 Regular languages

Let us start with discussing languages where membership can be decided with finite state automatas (DFAs).

4.2.1 Recapitulation: finite state automata

A deterministic finite state automaton (DFA) is a very simple computing device and we refer to Lecture 2 for a detailed discussion. For now, it is enough to remember that a DFA is characterized by a finite set of internal states and symbol-induced transitions between them. Formally, it corresponds to a 5-tuple $M = (Q, \Sigma, \delta, q_0, F)$, where Q is the finite set of internal states, Σ is the alphabet it can process and $\delta : Q \times \Sigma \to Q$ is a transition function. The remaining two objects are a designated starting state $q_0 \in Q$ and a set of accept states $F \subseteq Q$.

DFAs can process input strings $x = x_0 \cdots x_{n-1} \in \Sigma^*$ of arbitrary length $|x| = n$. But, they do this in a *static* – in the sense that a DFA reads input symbols one at a time – and *passive* – in the sense that it doesn't interact with the input – fashion.

4.2.2 Regular languages

Recall from Sec. 4.1.3 that entire classes of decision problems (yes/no questions) can be represented as languages $A \subset \Sigma^*$. Conversely, DFAs are designed to either accept or reject such strings. For a given and fixed DFA M, we define

$$L(M) = \{x \in \Sigma^* : \text{ DFA } M \text{ accepts input } x\} \subseteq \Sigma^*.$$

In words: this is the set of strings over Σ that is accepted by DFA M. Informally speaking, languages that can be decided by DFAs alone are among the 'easiest' computational problems conceivable. This class of problems deserves a proper name.

Definition 4.5 (regular language). A language $A \subseteq \Sigma^*$ is *regular* if there exists a DFA M such that $A = L(M)$. regular language

Example 4.6 The binary languages

$$\textsc{Parity} = \left\{x \in \{0,1\}^* : \text{ the last bit of } x \text{ is a } 1\right\} \quad \text{and}$$
$$\textsc{ParityOfSum} = \left\{x \in \{0,1\}^* : \ x \text{ contains an even number of 1s}\right\}$$

are both *regular languages*. After all, we constructed explicit DFAs that check the defining properties in Lecture 2 and Exercise Sheet I, respectively. ∎

4.2.3 Regular operations

Regular languages subsume, in a very precise and practical sense, the easiest types of computational problems. A single pass through the entire input, together with a finite amount of pre-specified transitions, suffices to decide membership unambiguously. This is a very desirable feature that is preserved by certain natural operations on sets (languages). The following rigorous statement asserts that certain combinations of regular languages again produce regular languages.

Theorem 4.7 (regular operations). Fix an alphabet Σ and let $A, B \subseteq \Sigma^*$ be *regular languages*. Then, the following three set combinations regular operations

$$A \cup B = \{x : \ x \in A \text{ or } x \in B\} \qquad \textit{(union)}, \tag{4.3}$$
$$AB = \{xy : \ x \in A \text{ and } y \in B\} \qquad \textit{(concatenation)}, \tag{4.4}$$
$$A^* = \{\varepsilon\} \cup A \cup AA \cup AAA \cup \cdots \qquad \textit{(star)}. \tag{4.5}$$

are also *regular languages*.

Proof. Part of Exercise Sheet II. ∎

Theorem 4.7 can be used to start with 'simple' regular languages and construct more involved, perhaps more interesting, ones. Here, we present one such construction that uses the *star* operation. To begin with, note that it is easy to design a DFA that only ever accepts two input bitstrings, namely

$x = 0$ and $x = 1$. We leave the precise construction as an exercise. According to Definition 4.5, this implies that the set $\{0, 1\}$ is a regular language. In turn, Theorem 4.7 ensures that the *star* of this language

$$\{0, 1\}^* = \{\varepsilon\} \cup \{0, 1\} \cup \{00, 01, 10, 11\} \cup \cdots = \{\varepsilon, 0, 1, 00, 01, 10, 11, \ldots\}$$

is also a regular language. This alternative construction of the set of all bitstrings $\{0, 1\}^*$ shows us where the star-superscript in Eq. (4.1) actually comes from. What is more, the language of all bitstrings is actually a regular language. This, however, may not be too surprising. After all, it is also very easy to write down a DFA that accepts every possible input.

Exercise 4.8 Construct a DFA over the binary alphabet $\Sigma = \{0, 1\}$ that only accepts 0 and 1. Also, construct a DFA over the binary alphabet that accepts every possible input string.

The three operations *union* (4.3), *concatenation* (4.4) and *star* (4.5) from Theorem 4.7 can be combined to show that other set-theoretic operations also preserve regular languages. This includes, in particular, *set intersection* and *complementation*.

Exercise 4.9 Let $A, B \subset \Sigma^*$ be two languages. Show that taking the *complement* complement

$$\overline{A} = \{x \in \Sigma^* : \ x \notin A\} = \Sigma^* \setminus A \subseteq \Sigma^*,$$

as well as set *intersection* intersection

$$A \cap B = \{x \in \Sigma^* : \ x \in A \text{ and } x \in B\} \subseteq \Sigma^*.$$

are also regular operations.

Example 4.10 The set of all (binary representations) of numbers divisible by 4 forms a regular language in $\{0, 1\}^*$.

To see this, note that it is easy to write down a DFA that checks if the last bit of a bitstring is zero (negate the parity-DFA from Lecture 2). Likewise it is easy to construct a DFA that checks if the second to last bit is zero. Hence, both $A = \overline{\textsc{Parity}} = \{x : \text{ last bit of } x \text{ is } 0\}$ and $B = \{x : \text{ second-to-last bit of } x \text{ is } 0\}$ are regular languages. The intersection of both regular languages is also regular and describes the set of all bitstrings where the last two bits are equal to zero. This is the case if and only if the corresponding number is divisible by 4. ∎

4.2.4 Fundamental limitations

DFAs and, by extension, regular languages are great, because they are easy and cheap to recognize. But, as we've already seen in Lecture 3, they do have their limitations. The following rigorous consequence of DFA computation turns out to be a versatile tool for identifying such limitations.

Lemma 4.11 ('Pumping lemma'). Let Σ be an alphabet and let $A \subseteq \Sigma^*$ be a *regular language*. Then, there exists a natural number $l > 0$, called the *pumping length*, that possesses the following property: We can decompose every string $x \in A$ with length $|x| \geq l$ into three substrings $x = uvw$ such that pumping lemma

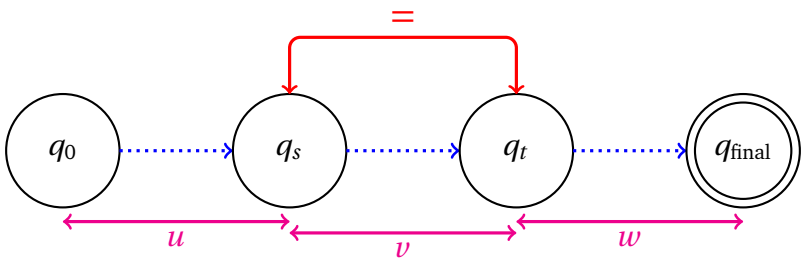


Figure 4.1 *Illustration of the proof idea behind Lemma 4.11:* Suppose that a DFA with n states processes an input string $x = uvw$ that contains strictly more than n symbols. Then, at least one internal state must be visited twice during the computation. Illustrated by the red equality bracket between q_s and q_t, this introduces a nontrivial loop in the computation. And, repeating it more than once must also produce a string that is accepted by the DFA. The magenta arrows at the bottom indicate that only the central portion v of the input string is affected by such repetitions.

1. $|v| \geq 1$ (the central substring is non-empty),
2. $|uv| \leq l$ (the length of the first two substrings doesn't exceed the pumping length),
3. $uv^k w \in A$ for all $k \geq 0$ (we can copy-paste the central substring as often as we like and still remain within the language).

This is a prime example of a 'Lemma' or 'Hilfssatz'. As a tool, it is remarkably powerful, because it rigorously allows us to prove that certain decision problems cannot be answered by a DFA – regardless how intricate and powerful its construction. But, by itself, it's not a catchy statement worthy of the title 'Theorem'. Here, we content ourselves with a proof sketch. For a detailed proof, we refer to standard material, e.g. [Wat20][Lecture 5] and [Sch20][Section 2.5].

Proof sketch for Lemma 4.11. By assumption, the language $A \subseteq \Sigma$ is regular. This means that there must exist a single DFA $M = (Q, \Sigma, \delta, q_0, F)$ that accepts a string x if $x \in A$ and rejects it otherwise ($x \notin A$). This DFA, however, can only have a finite number of, say, $|Q| = l$ internal states. And this can create problems if the string x to be processed is strictly longer than l. In order to accept a length-n string x, the DFA must transition through $n+1$ internal states $r_0 = q_0$, $r_{k+1} = \delta(r_k, x_k)$ and $r_{n+1} \in F$. Now, something interesting happens if n (the length of the string $x \in A$) is at least as large as l (the number of states): at least one of the internal states $q \in Q$ must be visited twice during the computation that ultimately accepts x. Visualized in Figure 4.1, this loop allows us to construct larger strings that must also be accepted by the DFA computation. Namely, we can repeat the central part of the string as often as we like. This is the content of the third (and most important) point in Lemma 4.11:

- The substring u comprises all symbols of x that are processed before the loop begins;

- the substring v contains symbols within the loop;
- the substring w comprises all symbols of x that are processed after the loop has ended.

The first two points are then easily established by careful bookkeeping. ■

Lemma 4.11 establishes a property that must always be true for regular languages. Once a string that belongs to the language becomes large enough (it has to be longer than the pumping length), we can start 'pumping' up the central part further and further without ever leaving the language. Hence, the name 'pumping lemma'. For certain languages, we can use pumping to deduce a contradiction. This mathematical contradiction can then only be resolved by conceding that the language in question cannot be a regular language to begin with (and, consequently, the pumping lemma need not apply in the first place).

Example 4.12 (SAME is not a regular language). The binary language

$$\textsc{Same} = \left\{1^n 0^n \in \{0,1\}^* : \ n \in \mathbb{N}\right\} \subset \{0,1\}^n \tag{4.6}$$

is *not* a regular language.

To see this, let us first assume that SAME were a regular language. Then, Lemma 4.11 must hold for a certain pumping length l. To arrive at a contradiction, we pick $x = 1^l 0^l$ which has length $|x| = 2l$ (twice the pumping length) and is clearly contained in SAME. The pumping lemma tells us that we can decompose this string into substrings $u, v, w \in \{0,1\}^*$ such that (0) $x = uvw$, (1) $|v| \geq 1$ and (2) $|uv| \leq l$. The last property tells us that uv cannot have any 0s in it. In fact, $v = 1^j$ for some $j \geq 1$, because $|v| \geq 1$. The final property of Lemma 4.11 (for $k = 2$) then implies that

$$uv^2w = 1^{l+j}0^l,$$

must also be part of the language SAME. Since this is obviously not the case ($j \geq 1$), we have arrived at a contradiction. The only way to resolve it is to concede that SAME cannot be a regular language to begin with. ■

This is an example of a mathematical proof by contradiction. Formally, they are as sound as constructive proofs (e.g. a row-reduced echelon form exists for every matrix, because we can write down an algorithm – Gauss-Jordan elimination – that does it) and counting arguments (e.g. there are $2^{n/2}$ palindromes among bitstrings of even length n), but conceptually they are often even more appealing. They sometimes allow us to mathematically prove that certain things cannot exist at all! Here is another example that brings our discussions from last lecture to a nice close.

proof by contradiction

Example 4.13 (PALINDROME is not a regular language). The language

$$\textsc{Palindrome} = \left\{x \in \Sigma^* : \ x^{\mathrm{R}} = x\right\}, \tag{4.7}$$

where the reverse operation R was introduced in Example 4.1, cannot be a regular language. The derivation is similar to Example 4.12 and we leave it as an exercise. ■

It is worthwhile to formulate the implications of Exercise 4.13 in a different fashion: *it is impossible to design a DFA that recognizes palindrome structure in bitstrings of arbitrarily large length*. This is the rigorous statement promised, but not derived, in Lecture 3.

PALINDROME cannot be decided with DFAs

4.3 (Semi-)decidable languages

4.3.1 Recapitulation: Turing machines

Turing machines (TMs) have been the core focus of Lecture 3. In a nutshell, a TM is a finite state automaton empowered by an additional memory tape and the ability to read/write, as well as moving the tape in both directions. Formally, a TM is a 7-tuple $M = \left(Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}}\right)$, where Q denotes the set of states in the finite state control mechanism, Σ is the input alphabet comprised of symbols to be processed and Γ is the tape alphabet that encompasses Σ as well as a designated blank symbol '$\sqcup$' to denote empty tape space. It is the transition function, where the deviation from ordinary DFAs becomes most apparent: $\delta : Q \times \Gamma \longrightarrow Q \times \Gamma \times \{L, R\}$ also allows to specify a write command (δ takes a tape symbol from Γ as input and spits out another tape symbol from Γ), as well as tape movement towards the left (L) or right (R). Finally, we must also specify an internal starting state (q_0), as well as two designated halting states ($q_{\text{accept}}, q_{\text{reject}}$) that prompt the TM to stop and output either accept or reject.

TMs can process input strings $x \in \Sigma^*$ of arbitrary length. And they do this in a *dynamic* – in the sense that the tape head can move back and forth on the working tape – and *active* – in the sense that a TM has read/write access to a working tape – fashion. They are also toy models for actual (electronic and human) computers. We will actually see later on that this model of computation is believed to be universal in some sense.

4.3.2 Decidable languages

Definition 4.14 (decidable languages). Fix an alphabet Σ and a language $A \subseteq \Sigma^*$. We say that A is *(Turing)-decidable* if there exists a Turing Machine (TM) M with the following two properties:

decidable languages

1. M accepts every string within the language ($x \in A$) and
2. M rejects every string not contained in the language ($x \in \bar{A} = \Sigma^* \setminus A$).

Note that both requirements have merit in their own right. In contrast to DFAs – where rejecting is the same as not accepting – Turing machines may loop on forever and not produce a definite yes/no answer at all. We will address this issue partly in the next subsection. For now, we point out that decidable languages are a strict generalization of regular languages.

Lemma 4.15 Every regular language is also a decidable language.

Proof. Let $A \subseteq \Sigma^*$ be a regular language. Then, there must exist a DFA that accepts every string within the language ($x \in A$) and rejects every string that

isn't ($x \in \bar{A}$). But, we can perfectly simulate this DFA with a (very stupid) Turing machine. ■

Lemma 4.15 formulates a rather boring relation between regular and decidable languages. What is more interesting, is the observation that the two language classes are actually distinct. Moving from DFAs to TMs does come with additional computing power.

Example 4.16 The language PALINDROME is a decidable language. After all, we have constructed a TM that accepts even-length bitstrings with palindrome structure and rejects all other even-length bitstrings. This TM construction can be generalized to handle even- and odd-length palindromes at once which establishes the claim. ▪

Together, Example 4.13 and Example 4.16 highlight that PALINDROME is a language that is decidable, but not regular.

Theorem 4.17 The set of all decidable languages is strictly larger than the set of all regular languages.

Or, in other words: Turing Machines are strictly more powerful than DFAs.

4.3.3 Semidecidable languages

It is now time, to take a serious drawback of Turing machines properly into account. Namely, that they may not stop at processing certain inputs and instead loop on forever without ever giving a accept/reject answer. Let M describe a Turing Machine over input alphabet Σ. We write

$$L(M) = \{x \in \Sigma^* : \text{ TM } M \text{ accepts input } x\} \subseteq \Sigma^*, \tag{4.8}$$

to collect all possible input strings that prompt the TM M to halt (at some point) and output the verdict 'accept'.

semidecidable languages

Definition 4.18 Fix an alphabet Σ and a language $A \subseteq \Sigma^*$. We say that A is *(Turing)-semidecidable* if there exists a Turing Machine (TM) M such that $A = L(M)$.

Lemma 4.19 Every decidable language is also semidecidable.

Proof. This is obvious, because Definition 4.18 follows from Definition 4.14 by dropping the second requirement (reject input strings that are not in the language). ■

What is slightly less obvious is that there are yes/no questions that are semidecidable, but not decidable. Here is a seemingly self-serving example that uses the fact that we can encode entire Turing Machines into bitstrings. (We will discuss this example and its striking consequences in detail in Lecture 5.)

Example 4.20 Define the (appropriately encoded) language

$$\text{ACCEPT} = \{(M, w) : \ M \text{ describes a TM, and } M \text{ accepts input } w.\}\,.$$

By construction, it is obvious that this language is *semidecidable* (run the TM M and see if it halts). However, if $(M, w) \notin$ ACCEPT, then there are two possibilities. Firstly, the TM may halt, but reject w. Alternatively, it may also not stop at all and instead loop on forever without producing an output at all. This starts to look scary. In fact, there is no way to make sure that another TM exists that detects the infinite loop and rejects the input. This will be the main scope of the next lecture. For now, we merely mention the final result (without proof): ACCEPT is semidecidable, but not decidable. ∎

interpreters can be troublesome

Note that Example 4.20 essentially describes an *interpreter*; that is a program which takes as input both a program (M), as well as an input (w) to that program, and simulates M on input w. Hence, our insight is rather discouraging. It looks as if it may actually be impossible to check for a finite runtime before starting to actually execute the interpreter. As we shall see in the next lecture, this is just the tip of a scary iceberg. For now, we report the immediate consequences.

Theorem 4.21 The set of all semidecidable languages is strictly larger than the set of all decidable languages.

4.3.4 Fundamental limitations

Given that TMs encompass a wide array of actual computing devices and semidecidability is a rather weak notion of problem solving, it seems plausible that the set of all semidecidable languages actually encompasses all possible languages conceivable. Surprisingly and shockingly, this is not true at all! On the contrary, almost all languages are not even semidecidable. This is the content of the following mathematically rigorous statement.

Theorem 4.22 There are (very many) languages $A \subseteq \Sigma^*$ that are *not* semidecidable (and therefore certainly not decidable, let alone regular).

proof by complete enumeration

The proof is based on *complete enumeration* which is, essentially, a more sophisticated counting argument.

Proof sketch. Fix an input alphabet Σ, as well as a tape alphabet $\Sigma \cup \{\square\} \subseteq \Gamma$. Then, every Turing machine that operates on this tape alphabet only has a finite number of degrees of freedom. We can vary the size of Q (the set of finite states), the starting state, the two halting states and the transition function $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$. But overall, the total number of choices is finite. By letting the number of internal states $|Q|$ grow gradually, we can keep track of all possible Turing machines by counting them and assigning a unique natural number $n \in \mathbb{N}$ to each of them. Granted, the total number of enumerated Turing machines does approach infinity as we let $|Q|$ get larger and larger. But, importantly, it does so in a controlled fashion. Much like the rational numbers $\mathbb{Q}$ (and the natural numbers $\mathbb{N}$ which can be used to enumerate all possible rational numbers), the total number of Turing machines is infinitely

large, but countable. This means that we can assign a unique number to each TM. And by letting the maximum number approach infinity, we can extend this enumeration process to *all* TMs conceivable.

In turn, every TM defines a language – namely the language $L(M)$ defined in Eq. (4.8). And, according to Definition 4.18, every semidecidable language $A \subseteq \Sigma^*$ must be of this form. But this implies that the total number of semidecidable languages can be at most as large as the total number of TMs (note that two or more TMs might give rise to the same language which is why the correspondence is not one-to-one) which is infinitely large, but countable. This allows us to conclude that the number of semidecidable languages is (at most) countably infinite.

But, here's the problem. The total set of all possible languages over Σ is *much* larger. It would correspond to the set of all subsets over Σ^* – which already is an infinite collection of strings. This set of all subsets would be manageable if Σ^* was finite to begin with. For finite sets S, the set of all subsets $\mathcal{P}(S)$ is called the power set. Its size grows exponentially with the number of elements in S, i.e. $|\mathcal{P}(S)| = 2^{|S|}$. Remarkably, this mismatch in size between S and $\mathcal{P}(S)$ extends to countably infinite sets as well. Cantor's theorem states that for any set S, the power set $\mathcal{P}(S)$ has a strictly larger cardinality. For the case at hand, we have $S = \Sigma^*$ – a countably infinite set. Cantor's theorem then implies that the cardinality of $\mathcal{P}(\Sigma^*)$ must be strictly larger. This is only possible if $\mathcal{P}(\Sigma^*)$ – the set of all languages over Σ^* – contains a number of elements (languages) that is uncountably infinite.

To complete the proof, we must simply observe that it is impossible to match elements from an uncountably infinite set (the set of all languages) with elements from a set that is only countably infinite (semidecidable languages). This follows from the very definition of uncountable infinity (an uncountable set is a set that contains too many elements to be countable).

■

You may have already seen a similar size discrepancy in one of your math lectures. The set of all natural (and even rational) numbers is countably infinite, but the set of all real-valued numbers is uncountably infinite. In fact, it is a very good exercise in mathematical reasoning and understanding to try to follow these arguments. The wikipedia page on Cantor's theorem, for instance, is very well-written and an ideal starting point. No strong mathematical background is required.

In the literature, semidecidable languages are sometimes also called *recursively enumerable* (RE) languages. The proof sketch above explains why.

semidecidable = recursively enumarable (RE)

As a conclusion, we summarize the mutual interrelations between the three discussed language classes in Figure 4.2

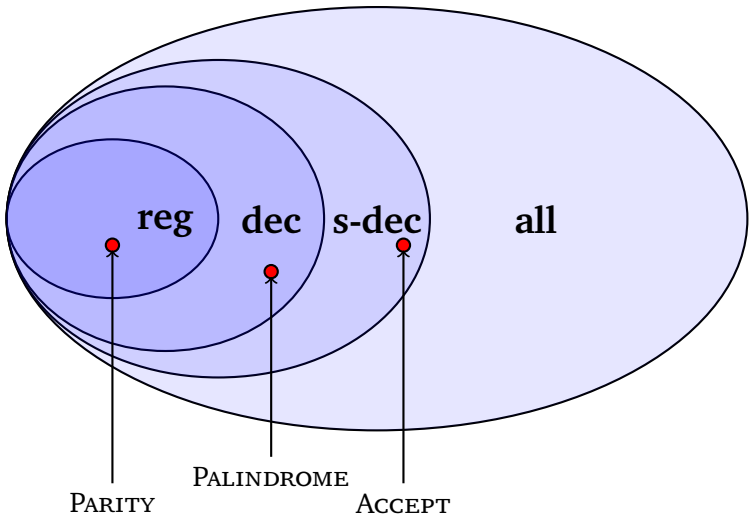


Figure 4.2 *Membership illustration among language classes:* regular languages (**reg**) are a subset of decidable languages (**dec**) which are, in turn, a subset of semidecidable languages (**s-dec**). Shockingly, semidecidable languages only encompass a small fraction of all languages (**all**). To underscore that these inclusions are strict, we also point out specific languages (decision problem) that belong to one language set, but not a subset thereof. E.g. PALINDROME is decidable, but not regular.

4.4 The Church-Turing thesis

Above we have seen that the class of all decision problems (languages) that can be solved by Turing machines (TMs) seems painfully restrictive. They only cover a vanishingly small fraction of all decisions problems that are conceivable. How should we deal with this information?

A natural step forward would be to ditch the TM model of computation and replace it with another computational primitive that is strictly more powerful. After all, this is precisely how we moved from studying DFAs to studying Turing machines in the first place. Alas, such an 'update' may not be possible, because it is widely believed that the TM model already is (equivalent to) the most powerful computing process out there. This is the content of the following statement named after two scientists that developed different, but equivalent, universal models of computation in the 1930s.

Computational Primitive **(Church-Turing thesis).** Any function that can be computed by a mechanical (or physical) process can also be computed by a Turing machine.

Church-Turing thesis

Note that this is not a mathematical statement that can be proved or disproved. Instead it is part of a belief system that is strong within the (computer) science community. This also includes quantum computer scientists. The quantum computer has no effect on such a grand scale of difficulty settings:

every problem that is (semi-)decidable with a quantum computer is also (semi-)decidable with a TM and vice versa.

However, there is a stronger notion of the Church-Turing thesis – called the *strong Church Turing thesis* – that seems to be refuted by the existence of quantum computers. But, this is a question about computational efficiency, not computational possibility, and therefore a topic for a later lecture.

Bibliography

[Sch20] W. Schreiner. *Computability and Complexity*. JKU Linz, Austria, 2020. URL: https://moodle.risc.jku.at/pluginfile.php/9330/course/section/1427/main.pdf?time=1594822187866.

[Sip97] M. Sipser. *Introduction to the theory of computation*. PWS Publishing Company, 1997. ISBN: 978-0-534-94728-6.

[Wat20] J. Watrous. *Introduction to the Theory of Computing (lecture notes)*. University of Waterloo, Canada, 2020. URL: https://cs.uwaterloo.ca/~watrous/ToC-notes/ToC-notes.03.pdf.