Introduction to Computational Complexity

(Berechenbarkeit und Komplexität)

Johannes Kepler University Linz / Fall 2023

Assoc. Univ.-Prof. Dr. Richard Kueng

Date: Fall 2023

Contents

1. Motivation and (some) background

Date: October 5, 2023

Agenda:

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

1.1 Motivating example: traveling salesperson (TSP)

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

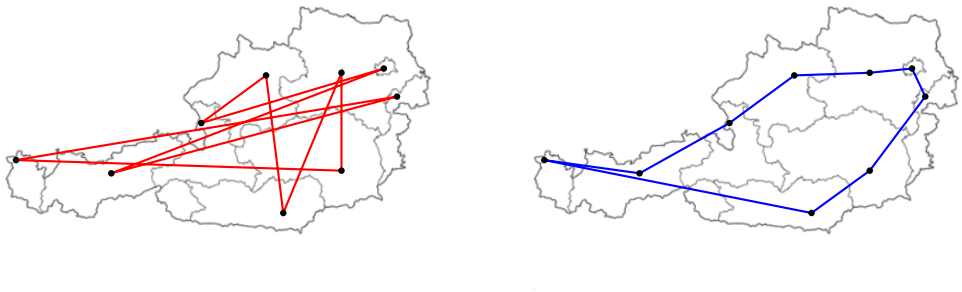


Figure 1.1 *Illustration of the traveling salesperson problem (TSP)*: A traveling salesperson intends to visit all Austrian state capitals (black dots) exactly once during a business trip. After the trip is completed, she also wants to return to the origin city. The goal is to find the shortest possible 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. For Germany (16 state capitals), this number grows to is 120, 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 faster than any exponential). 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 $\approx 9! \times 10^{-3} \approx 6$min, which is still doable. But the German TSP (16 state capitals) would already require $16! \times 10^{-3}$s ≈ 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, 13 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 Space complexity
10 **co-NP** and the polynomial hierarchy
11 Circuits
12 Circuit size-bounded computations
13 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 – 10), 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 Florian Schwarcz)
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.

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.

bit encoding

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 12, 2023

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 computations
3. Nondeterministic finite automata (NFAs)
   (i) determinism vs. nondeterminism
   equivalence relations (no 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 deserves[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 where 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]*.

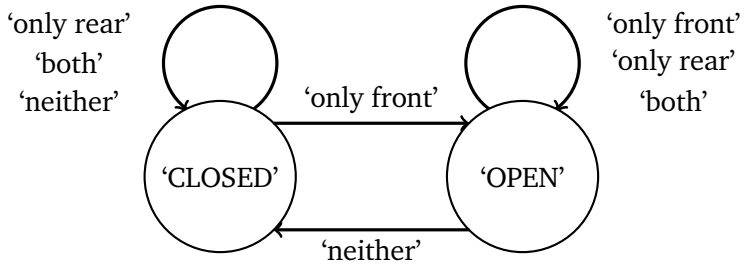


Figure 2.2 *State diagram describing an automatic door controller:* circles denote different states the door can be in ('OPEN' or 'CLOSED'). Possible transitions between states are depicted by directed arrows. E.g. the trigger 'only front' makes the door change states from 'CLOSED' to 'OPEN'.

	'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 later on.

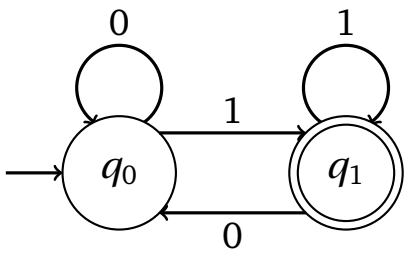


Figure 2.3 *State diagram for a simple machine that processes bitstrings:* The two possible internal states are denoted by circles, directed arrows between the two label transitions. The single arrow coming from the left marks the starting state (q_0 in this case). The 'double circle' singles out final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

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 terminate and report 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)):

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

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 natural numbers that are odd. 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*.

[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.

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

- 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

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

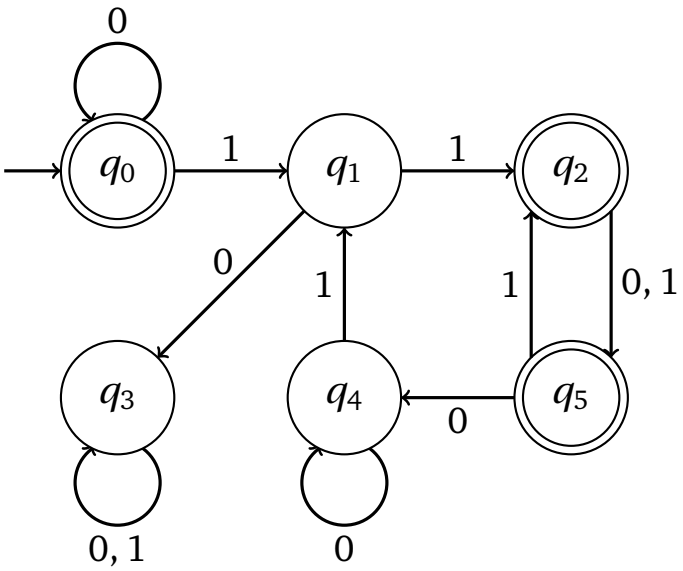


Figure 2.4 *Example of a more involved DFA state diagram.*

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

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

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

	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).

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

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 next 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 a choice

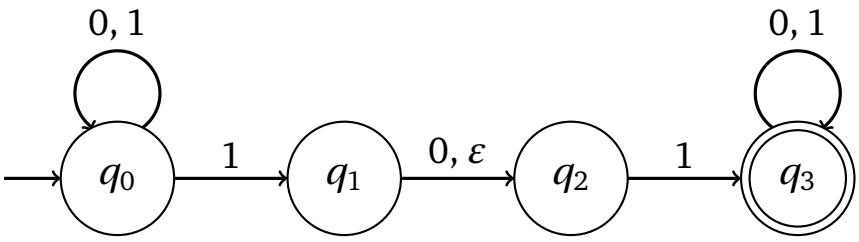


Figure 2.5 *State diagram of a NFA:* this state diagram violates the rules of a DFA. In particular, there are two 1-arrows at state q_0 (leftmost circle). The state q_1 has one exiting arrow for 0, but note for 1. The ε-arrow (empty string) between q_1 and q_2 indicates that such a transition is possible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

(that is hidden from us): it can choose which arrow to follow. The absence of an arrow instead prevents the computation from proceeding.

(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}.

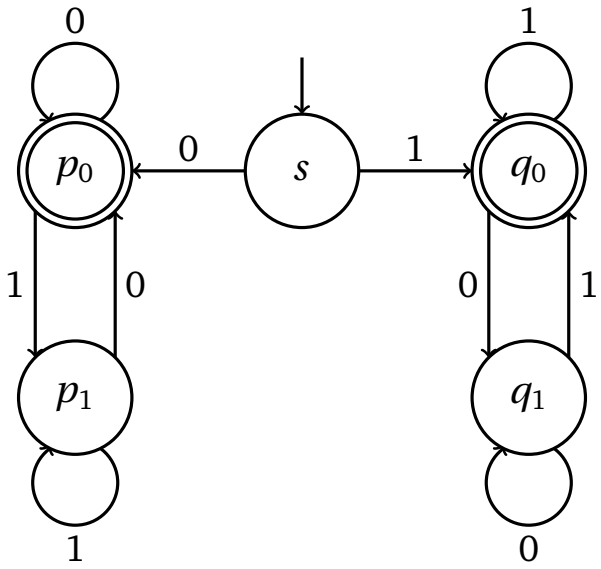
1 Draw a state diagram that characterizes this DFA (it should contain two circles and 2 outgoing arrows for each of them).
2 Convert this state diagram into a formal description according to Definition 2.3. Specify the transition function by completing the following table and inserting the correct states:

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

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

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

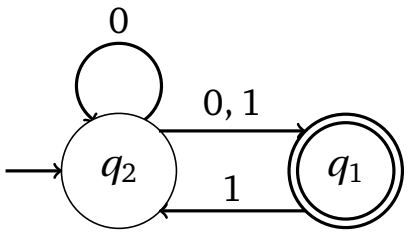
1 Draw the associated state diagram.
2 Which one of the following statements is true:

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 19, 2023

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 is one topic of Exercise Sheet I. 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. 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 $2^{n/2}$ distinct internal

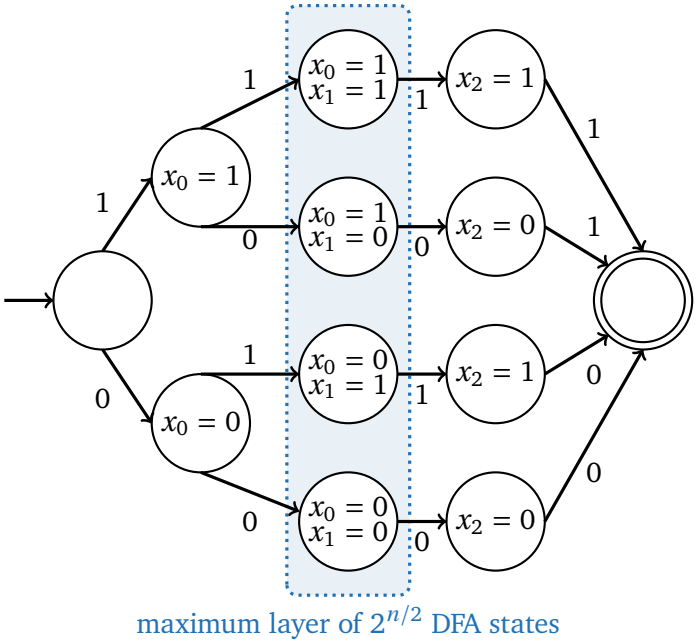


Figure 3.1 *Illustration of a DFA that accepts palindromes of length $n = 4$:* transition arrows not shown here would lead to a rejection of the palindrome property. This could be implemented, for instance, by self loops in the second and third layer.

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 the proof idea in a future lecture.

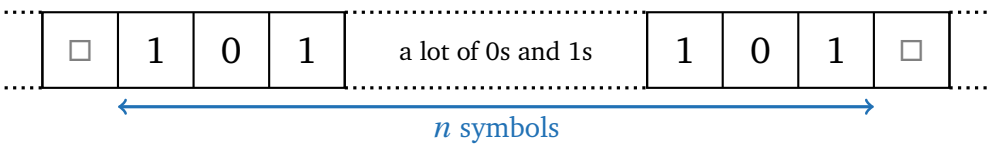


Figure 3.2 *Setup for identifying (even) palindromes:* We are given a bitstring x of even length n and must check whether it is a palindrome or not. We envision that the bitstring in question is written onto a very long, checkered strip of paper. The square symbol (□) denotes empty paper space at either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

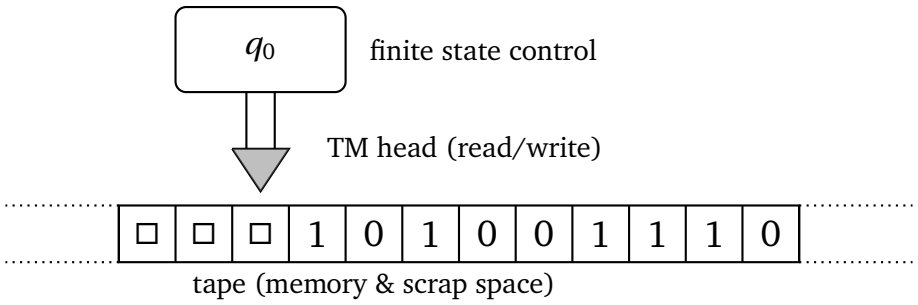


Figure 3.3 *Schematic illustration of a TM.*

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

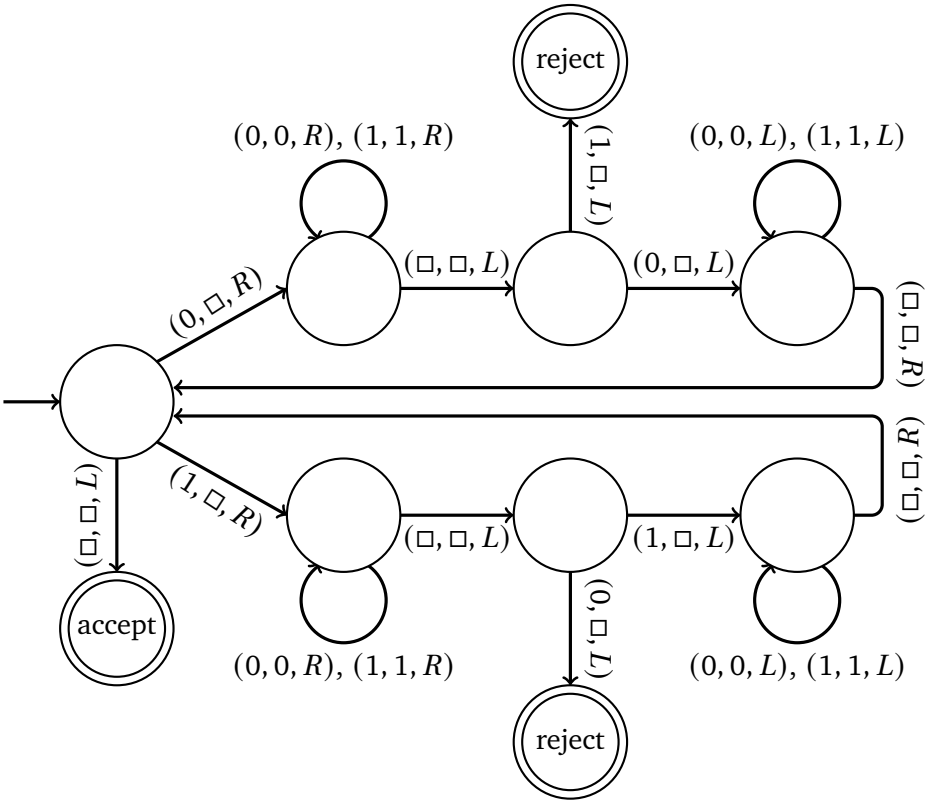
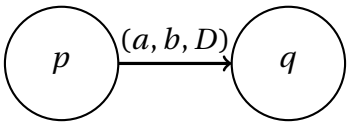


Figure 3.4 *State diagram for the palindrome checking Turing machine (even length) from Section 3.3.* We assume that we already start at the location of the very first bit. If it is 0, we enter into the upstairs trajectory. If it is 1, we enter the downstairs trajectory instead. Either trajectory can 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 algorithm. 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 the father of both artificial intelligence and theoretical computer science. 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

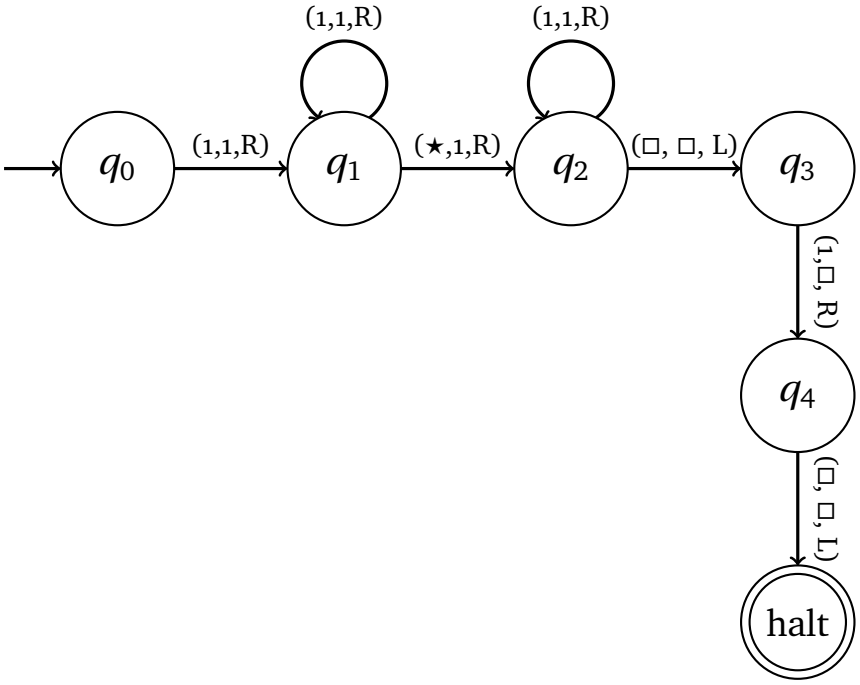
Problem 3.10 (state diagram to Turing machine I). We consider Turing machines over the alphabet $\Sigma = \{1, \square, *, \star\}$, where (i) 1-symbols are used to write inputs & outputs, $\square$ denotes 'empty' tape space, $*$ is used to cross out input symbols that have already been processed and $\star$ 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?

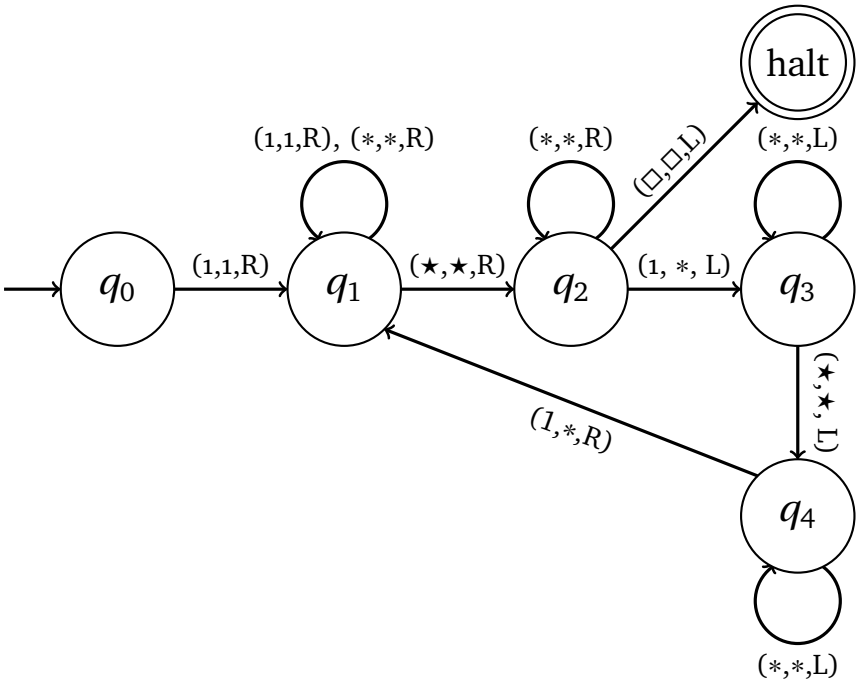
- **a)** (★ = +) the TM computes the *sum* of two numbers (in unary encoding) e.g. input $\underline{1}$ 1 1 ★ 1 1 □ produces output 1 1 1 1 1 $\underline{□}$ □.
- **b)** (★ = −) the TM computes the *difference* of two numbers (in unary encoding), e.g. input $\underline{1}$ 1 1 ★ 1 1 □ produces output 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: November 3, 2023

Agenda

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 Turing machines. Perhaps surprisingly, we will see that there are also limitations. And shockingly, there might be no conceivable way to overcome them.

Agenda:

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

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,

decision problem

but today we formulate it in a slightly different fashion. Define the *reverse 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 automata.

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.

Finally, we emphasize that our restriction to deterministic finite automata is without loss. This is because it is possible to convert nondeterministic finite state automata into functionally equivalent DFAs, see Lecture 2.

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 bigger, 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 accepts two input bitstrings of length 1, 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 $A = \overline{\text{PARITY}} = \{x : \text{ last bit of } x \text{ is } 0\}$ 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.

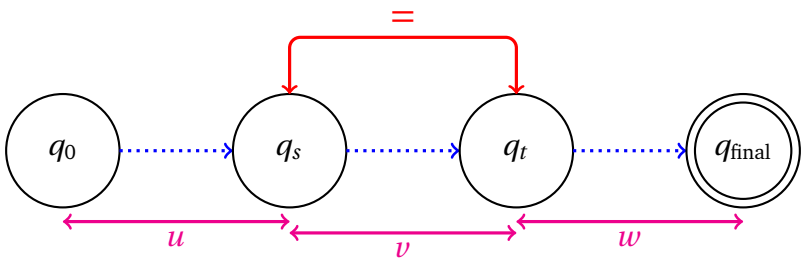


Figure 4.1 *Illustration of the proof idea behind Lemma 4.11:* Suppose that a DFA with n states processes an input string $x = uvw$ that contains strictly more than n symbols. Then, at least one internal state must be visited twice during the computation. Illustrated by the red equality bracket between 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.

pumping lemma

Lemma 4.11 ('Pumping lemma'). Let Σ be an alphabet and let $A \subseteq \Sigma^*$ be a *regular language*. 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

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).

It 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) $v \neq \varepsilon$ 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 $v \neq \varepsilon$. 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. ■

proof by contradiction

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.

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 2.

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: $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$ 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'.

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 $A = L(M)$. semidecidable languages

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

$$\textsc{Accept} = \{\langle M, w\rangle : \ M \text{ describes a TM, and } M \text{ accepts input } w.\}\,.$$

By construction, it is obvious that this language is *semidecidable* (run the TM M and see if it halts). However, if $\langle M, w\rangle \notin \textsc{Accept}$, 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 the example essentially describes an *interpreter*; that is a program which takes as input both a program (M) and an input (w) to that program, and simulates M on input w. The result 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 seem to 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 number theory. 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 *completely enumerable* (CE) languages. The proof sketch above explains why.

semidecidable = completely enumarable (CE)

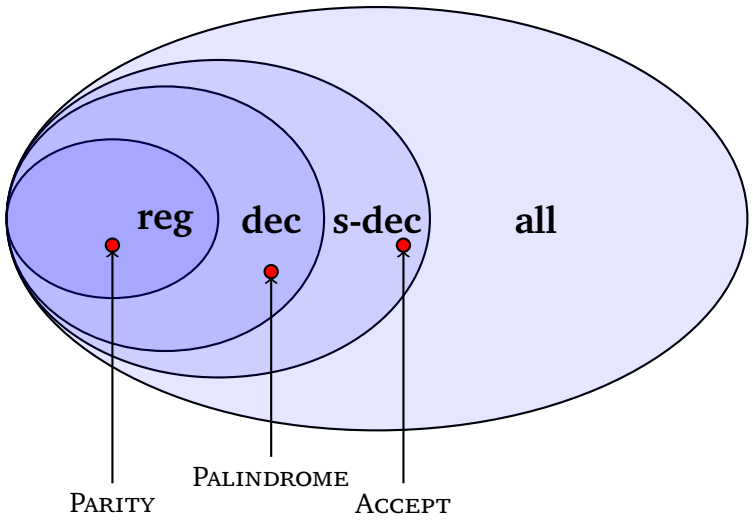


Figure 4.2 *Membership illustration among language classes:* regular languages (**reg**) are a subset of decidable languages (**dec**) which are, in turn, a subset of semidecidable languages (**s-dec**). Shockingly, semidecidable languages only encompass a small fraction of all languages (**all**). To 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.

As a conclusion, we summarize the mutual interrelations between the three discussed language classes in Figure 5.1

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 seems painfully restrictive. They only cover a vanishingly small fraction of all possible language problems. How should we deal with this information?

A natural step forward would be to ditch the Turing machine 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.

5. Universal Turing machines and undecidability

Date: 9 November 2023

Agenda

Today, we will complete our analysis of languages and their intrinsic difficulty. Last week we have seen that regular languages – i.e. decision problems that can be solved by a (deterministic) finite state automaton – are a strict subset of decidable languages – i.e. decision problems that can be solved by Turing machines (TMs). We have also shown that this inclusion is strict by presenting some languages (SAME and PALINDROME) that are decidable, but cannot be regular. Today, we will focus on the class of semidecidable languages – i.e. decision problems where we can identify yes-instances with a TM, but may not be able to say anything about no-instances – and the set of all remaining languages. In analogy to last lecture, we will identify some languages (ACCEPT and HALT) that are semidecidable, but cannot be decidable. We will also show that the complement of these languages (co-ACCEPT and co-HALT) are not even semidecidable. See Figure 5.1 for a more complete illustration of different language classes and their interdependencies.

The realization that some decision problems are *uncomputable* – in the sense that they are not even semidecidable – has profound implications for computer science, formal logic, mathematics and philosophy.

Agenda:

1. Encoding Turing Machines into bitstrings
2. Universal Turing Machines
3. Undecidable & uncomputable problems
4. Interpretations and implications

5.1 Bit encoding of Turing machines

5.1.1 Encoding tuples into bitstrings

Before moving on to Turing machines, let us start with an easier warm-up: how do we encode an ordered list (x, y) of two numbers $x, y \in \mathbb{N}$ into a bitstring?

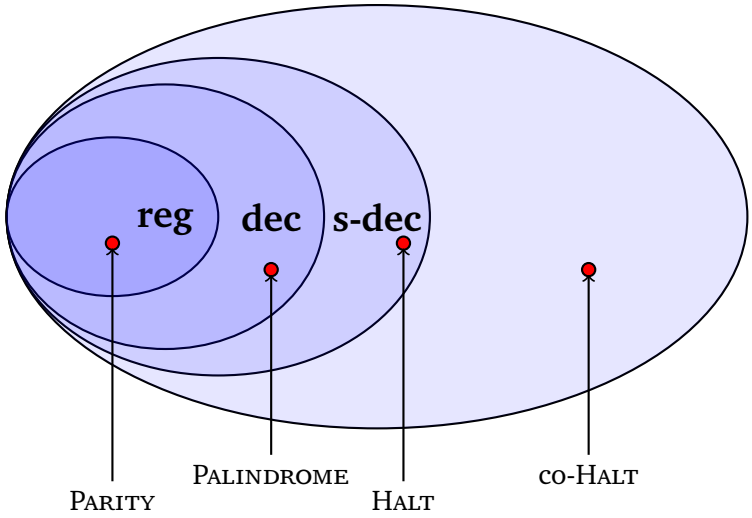


Figure 5.1 *Different language classes with exemplars (continuation from Lecture 4):* regular languages (**reg**) are a subset of decidable languages (**dec**) which are a subset of semidecidable languages (**s-dec**) which, in turn, are a subset of all languages. The halting problem (Halt) is semidecidable, but not decidable. Its complement co-Halt is not even semidecidable.

In this particular case, two steps suffice. Firstly, we must find a way to represent individual constituents as bitstrings. For this example, this is easy. We can simply use bit encodings of natural numbers: $x \mapsto \llcorner x \lrcorner \in \{0,1\}^*$ and $y \mapsto \llcorner y \lrcorner \in \{0,1\}^*$. But a simple concatenated string $\llcorner x \lrcorner \, \llcorner y \lrcorner$ is not good enough, because it doesn't tell us where the first bit encoding ends and where the second one starts. This issue can be resolved by introducing an additional *roadblock symbol*, say # and writing down $\llcorner x \lrcorner \# \llcorner y \lrcorner$ in the ternary alphabet $\{0,1,\#\}$. Subsequently, we can encode this alphabet into bitstrings. There are many ways to do so. Here is a particularly simple encoding that increases bit-length by a factor of two:

roadblock symbol #

$$0 \mapsto 00, \;\; 1 \mapsto 11, \;\; \# \mapsto 01.$$

Putting everything together, we obtain a bit encoding

$$(x, y) \mapsto \llcorner (\llcorner x \lrcorner \# \llcorner y \lrcorner) \lrcorner \in \{0,1\}^*,$$

whose encoding length $l = |\llcorner (\llcorner x \lrcorner \# \llcorner y \lrcorner) \lrcorner|$ obeys

$$\begin{aligned} l =& 2\,(|\llcorner x \lrcorner| + |\llcorner y \lrcorner| + 1) \\ =& 2\,(\lfloor \log_2(x) \rfloor + \lfloor \log_2(y) \rfloor + 3) \\ \leq& 2 \log_2(xy) + 6. \end{aligned}$$

Importantly, this encoding strategy readily extends to ordered tuples comprised of more than two elements:

bit encoding of tuples

$$(t_0, \ldots, t_{N-1}) \mapsto \llcorner (\llcorner t_0 \lrcorner \# \llcorner t_1 \lrcorner \# \cdots \# \llcorner t_{N-1} \lrcorner) \lrcorner \in \{0,1\}^*.$$

Even better, the individual tuple elements t_i need not be natural numbers to begin with. All that matters is that we can specify suitable bit encodings $t_i \mapsto \llcorner t_i \lrcorner \in \{0,1\}^*$ for each of them. Such bit encodings exist for ordered lists of symbols $\{0,\ldots,K-1\}$ (simply encode the number of elements), approximations of real-valued numbers (encode a floating point representation), vectors (encode a 1D-array), tables and matrices (encode a 2D-array), etc. It is also possible to check that such an encoding scheme assigns exactly one bitstring to a given tuple. This is essential, as it allows us to exactly reconstruct the original tuple from its bit encoding.

Exercise 5.1 (Bit encoding for TSP). Fix a number of cities $n \in \mathbb{N}$. Let $T \in \mathbb{R}^{n\times n}$ denote a pairwise distance table, i.e. a $n \times n$ table filled with nonnegative (floating point) numbers $t_{ij} \in \mathbb{R}$, $1 \leq i,j \leq n$. Also, let $d \in \mathbb{R}$ $(d > 0)$ be a kilometer count. Show that it is possible to encode the 2-tuple (T,d) into a single bitstring $\llcorner(\llcorner T \lrcorner, \llcorner d \lrcorner)\lrcorner \in \{0,1\}^*$. **Context:** This transformation encodes the TSP-related decision problem *"does the pairwise distance map T permit a TSP route that requires at most d kilometers"* into a single bitstring.

5.1.2 Encoding Turing machines into bitstrings

We have just seen that we can encode general tuples into bitstrings. This, in turn, also allows us to encode an entire Turing machine (TM). After all, TMs are formally just 7-tuples

$$M = \left(Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}}\right). \tag{5.1}$$

This is an important conceptual observation, and it makes sense to discuss possible encodings in slightly more detail.

The first three constituents in Eq. (5.1) (Q, Σ and Γ) all describe finite sets of symbols. Each of them can be specified by simply storing their total number. For instance, if $Q = \{q_0,\ldots,q_{N-1}\}$ has $|Q|$ elements, we can simply store $\llcorner |Q| \lrcorner$ and use bitstrings of length $\lfloor \log_2(|Q|)\rfloor + 1$ to enumerate all possible internal states. Naturally, we can do the same to also specify input alphabet Σ and tape alphabet Γ.

Now, the last three tuple elements $q_0, q_{\text{accept}}, q_{\text{reject}}$ can be specified by remembering which of the $|Q|$ states are special. E.g. if q_0 is the first state of Q, then we would write $\llcorner q_0 \lrcorner = 0\cdots 0$ to encode that q_0 it is the first element of Q. And we can store similar bitstring encodings of q_{accept} and q_{reject}.

The transition function is where things get a bit more interesting. After all, we need to encode an entire function

$$\delta : Q \setminus \left\{q_{\text{accept}}, q_{\text{reject}}\right\} \times \Gamma \to Q \times \Gamma \times \{L,R\}.$$

This function, however, is completely specified by a total of $(|Q|-2)\times|\Gamma|$ equations:

$$\delta(p,a) = (q,b,D) \quad \text{where} \quad p,q \in Q,\ a,b \in \Gamma,\ D \in \{L,R\}. \tag{5.2}$$

The number $(|Q|-2)\times|\Gamma|$ counts the total number of possible input combinations $(p,a)\in Q\setminus\{q_{\text{accept}},q_{\text{reject}}\}\times\Gamma$ and we can use tuple encodings to represent each transition equation (5.2) by a bit encoding

$$\llcorner(\llcorner p\lrcorner\#\llcorner a\lrcorner\#\llcorner q\lrcorner\#\llcorner b\lrcorner\#\llcorner D\lrcorner)\lrcorner\in\{0,1\}^*.$$

Subsequently, we can subsume all such bit encodings into an ordered list of $(|Q|-2)\times|\Gamma|$ bitstrings that fully specifies the transition function. Putting everything together provides us with a bit encoding of an entire TM:

$$\llcorner M\lrcorner=\llcorner\big(\llcorner Q\lrcorner\#\llcorner\Sigma\lrcorner\#\llcorner\Gamma\lrcorner\#\llcorner\delta\lrcorner\#\llcorner q_0\lrcorner\#\llcorner q_{\text{accept}}\lrcorner\#\llcorner q_{\text{reject}}\lrcorner\big)\lrcorner\in\{0,1\}^*.$$

We can use this encoding strategy for all conceivable TMs. Doing so ensures that every TM is represented by a single bit encoding (injectivity). But, conversely, not every bitstring will provide a valid encoding of a TM. It will be convenient if we adjust this stringent encoding a little bit to get a nicer correspondence between TMs and bitstrings. In particular, there are reasons to want that

1. every bitstring in $\{0,1\}^*$ represents *some* TM;
2. every TM is represented by infinitely many strings.

The first property is easy to ensure by associating strings that are not valid TM encodings with some canonical TM, e.g. the TM that immediately halts and outputs reject for any input. The second property can be ensured by tampering with our encoding scheme. In particular, we can allow that our bit representation can end with an encoded roadblock symbol #, followed by an arbitrary number of 1s that are ignored (think of comments in programming languages that also allow for adding superfluous symbols to any given program). Finally, recall that there is a one-to-one correspondence between natural numbers $n\in\mathbb{N}$ and bitstrings $\llcorner n\lrcorner\in\{0,1\}^*$. This allows us to identify TMs M with natural numbers n (by equating the bit encodings of both).

Turing number

Proposition 5.2 (Turing number). There is a many-to-one correspondence between natural numbers and Turing machines such that (i) the bit encoding $\llcorner n\lrcorner\in\{0,1\}^*$ of each n encodes exactly one TM M and (ii) every TM M is represented by (bit encodings of) infinitely many natural numbers.
We write $\llcorner M\lrcorner=n$ and call n a *Turing number* for TM M.

This correspondence between TMs and natural numbers also highlights that the total number of TMs is countable, because the total number of natural numbers is countable. We made heavy use of this observation in the proof of Theorem 4.22 (Lecture 4), when we showed that the total number of languages (which is uncountably infinite) is *much* larger than the total number of semidecidable languages (which is only countably infinite).

5.2 Universal Turing machines

Turing numbers are not only relevant for carrying out abstract counting arguments. They also provide a concrete way of specifying TMs in a machine-readable fashion. And this opens the door for feeding a bit encoding of one TM

as (part of an) input into another TM. In fact, it is possible to devise a *universal Turing machine* that is capable of simulating the execution of any other TM M for any possible input x.

This universality is another parallel between TMs, on the one hand, and actual computers on the other. Computers are also universal in the sense that they can simulate every other (reasonable) computing device. And it is this feature that makes them strictly more powerful and interesting than special-purpose devices, like calculators (or DFAs).

For simplicity and concreteness, we now restrict our attention to TMs M that work over the binary input alphabet $\Sigma = \{0,1\}$. In turn, the possible inputs for M are bitstrings $x \in \{0,1\}^*$. This is not a severe restriction, because we can simulate TMs with different input alphabets by another functionally equivalent TM that works over 0 and 1 only. Such a reduction is part of Exercise Sheet II.

Theorem 5.3 (Universal Turing machine). There exists a TM U such that for every $\llcorner n \lrcorner, w \in \{0,1\}^*$, $U(n,w) = M_n(w)$, where M_n denotes the TM represented by Turing number n. Moreover, if M_n halts on input w within T steps, then $U(n,w)$ also halts within a number of steps proportional to T^2.

universal Turing machine

A more careful argument shows that the runtime can actually be bounded by $CT\log(T)$, where C is a constant that only depends on M_n's alphabet size and the number of internal states. We refer to [AB09][Sec. 1.A] for a detailed argument on how to achieve this improved scaling bound.

To keep things nice and clean, we also assume that the TM M to be simulated actually has three tapes: one input tape, one working tape and one output tape. Each tape has its own tape head and can be moved independently. However, the input tape is only used for reading the input string x and the output tape is only used for writing down outputs. The following fact shows that this reformulation is not severe.

Fact 5.4 (simulating multi-tape TMs). Let $\tilde{M}$ be a TM that uses k independent tapes and runs for T steps. Then, we can simulate this computing process by a single-tape TM M that runs for (at most) $5kT^2$ steps. ∎

We leave the proof as an instructive exercise and move on to provide a high-level proof argument for Theorem 5.3 instead.

Proof sketch for Theorem 5.3. We explicitly construct an intermediate universal TM $\tilde{U}$ that uses the tape alphabet $\Gamma = \{0,1,\square\}$ and three working tapes in addition to its input and output tape. So, there will be 5 tapes in total. The TM $\tilde{U}$ will use one of its work tapes to simulate the work tape of TM M. It will use one additional working tape to store the transition function δ of M (in the form of an encoded transition table) and another work tape to store the current internal state of M.

To simulate one computational step of M, the TM $\tilde{U}$ scans the encoded

transition table and encoded current state of M to find the new state, tape symbols to be written, as well as head movement. It then executes all of them on the simulated work tape of M. We see that $\tilde{U}$ requires a total of C steps for each simulated computing step of M, where C is some number that only depends on the size of the encoded transition function $\langle\delta\rangle$.

Finally, we use Fact 5.4 to simulate the 5-tape TM $\tilde{U}$ by another TM U that uses only one tape. This, however, does introduce a quadratic slowdown:

$$\begin{array}{ccccc} M & \Rightarrow & \tilde{U} & \Rightarrow & U, \\ T & \rightarrow & CT & \rightarrow & 25C^2T^2. \end{array}$$

■

5.3 Uncomputable & undecidable decision problems

5.3.1 Variants of halting problems

We have seen that Turing machines are a universal model of computation. That is, we can use a certain TM – the universal TM U from Theorem 5.3 – to simulate *any* other TM. And, by construction, TMs are designed to answer decision problems. A very natural and useful decision problem is: does TM M (think: algorithm or program or software package) produce a reasonable output if we run it on input w? If we recast this yes/no question as a language over $\Sigma = \{0,1\}$, we obtain

$$\textsc{Accept} = \{\llcorner(M,w)\lrcorner : \ M \text{ is a TM and accepts } w\} \subset \{0,1\}^* . \tag{5.3}$$

Recall from Lecture 4 that a language A is called semidecidable if there exists a TM that accepts strings belonging to A. (But, it does not need to reject strings that don't belong to A).

Lemma 5.5 The language Accept introduced in Eq. (5.3) is semidecidable.

Proof. Use the universal TM U from Theorem 5.3. Accept a given input $\llcorner(M,w)\lrcorner$ if $U(\llcorner(M,w)\lrcorner) = M(w) =$ accept. ■

Note that it has been extremely convenient, that semidecidable languages are only concerned with yes/accept instances. This has allowed us to ignore cases where the encoded TM M may not halt at all on input w. This desirable feature seems to get lost if we ask the complementary question: does TM M (think: algorithm or program or software package) not produce a reasonable output if we run it on input w? Formally speaking, the *complement* of a language $A \subseteq \Sigma^*$ is comprised of all strings that do not belong to the language:

$$\text{co-}A = \overline{A} = \Sigma^* \setminus A = \{x \in \Sigma^* : \ x \notin A\} \subseteq \Sigma^*.$$

In theoretical computer science, we often use the prefix '*co-*' to denote the complement of something. The complement of the language Accept is.

complement co-A of a language A

$$\begin{aligned} \text{co-}\textsc{Accept} &= \{0,1\}^* \setminus \textsc{Accept} \\ &= \{\llcorner(M,w)\lrcorner : \ M \text{ is a TM that doesn't accept } w\}, \end{aligned} \tag{5.4}$$

which looks scarier than ACCEPT. Here, we have used the fact that our Turing number construction assigns a TM to every possible bitstring. This prevents us from having to worry about bitstrings that do not describe valid TM encodings in the first place.

In particular, co-ACCEPT only accepts inputs that encode a TM M that rejects or doesn't halt on input w. And it is not clear if we can use simulation on a universal TM at all to answer this question. Suppose, for example, that we have already simulated $M(w)$ for a *very* long time – say, 2.4×10^{67} computational steps – and it hasn't halted yet. Then, we are still non the wiser. After all, we still can't exclude the possibility that $M(w)$ will eventually halt and accept. The only possibility to really disprove this possibility is to let the simulation continue forever.

Another related yes/no question is the so-called *halting problem*. There, we drop the distinction between accepting or rejecting an input and simply ask if a TM M halts on input w at all. Reformulated as a language over bitstrings, the halting problem becomes

halting problem

$$\text{HALT} = \{\llcorner(M,w)\lrcorner : \ M \text{ is a TM that halts on input } w\} \subset \{0,1\}^* . \quad (5.5)$$

Note that ACCEPT $\subseteq$ HALT, because every accepting computation must also halt at some point. This suggests that checking yes-instances of HALT may be harder than checking yes-instances of ACCEPT only. Below, we will see (and exploit) that this intuition is true. The halting problem is, however, also semidecidable.

Lemma 5.6 The language HALT introduced in Eq. (13.1) is semidecidable.

The proof is almost identical to the proof of Lemma 5.5, so let us move on to the complementary language

$$\begin{aligned}\text{co-HALT} &= \{0,1\}^* \setminus \text{HALT} \\ &= \{\llcorner(M,w)\lrcorner : \ M \text{ is a TM that doesn't halt on input } w\} . \end{aligned} \quad (5.6)$$

This language looks even nastier than co-ACCEPT above. In fact, just looking at the definition should give us the chills. It is not obvious at all how we should (semi-)decide this language at all.

In the remainder of this lecture, we use mathematical reasoning to rigorously prove that the two languages co-ACCEPT and co-HALT are among the most difficult decision problems conceivable. We will show that they are *uncomputable* in the sense that they are not even semidecidable.

Warning 5.7 The fact that a language, like co-HALT, is uncomputable, does not mean that every problem instance is difficult. On the contrary, there are (infinitely) many Turing numbers that describe very simple TMs M with exceedingly benign behavior. E.g. our palindrome TM from Lecture 3 is certainly one of them. The main results of this lecture merely highlight that

there must be (at least) some nasty inputs, where we cannot hope to get a reasonable answer. ∎

5.3.2 Two undecidable languages

We say that a language is *undecidable* if it is not decidable. undecidable = not decidable

Theorem 5.8 The language ACCEPT introduced in Eq. (13.1) is undecidable.

This is a statement about the non-existence of something and we prove it by establishing a contradiction.

Proof. Assume (for the sake of deducing a contradiction) that ACCEPT is decidable. Then, there must exist a TM A such that

$$A\left(\llcorner(M,w)\lrcorner\right) = \begin{cases} \text{accept} & \text{if } M \text{ accepts } w, \\ \text{reject} & \text{else if } M \text{ does not accept } w, \end{cases}$$

for every possible input tuple (M, w). Now, we embed A as a subroutine into another TM. This TM, let's call it D, queries A to explore how a TM M would behave if we let it run on its own Turing number $\llcorner M \lrcorner$ (encoded as a bitstring). Once D has determined what would happen, it does exactly the opposite:

$$D\left(\llcorner M \lrcorner\right) = \neg A\left(\llcorner(M, \llcorner M \lrcorner)\lrcorner\right) = \begin{cases} \text{accept} & \text{if } M \text{ does not accept } \llcorner M \lrcorner, \\ \text{reject} & \text{else if } M \text{ does accept } \llcorner M \lrcorner. \end{cases}$$

Access to the TM A as subroutine allows TM D to process arbitrary Turing numbers. This, however, becomes problematic if we run D on the Turing number of itself:

$$D\left(\llcorner D \lrcorner\right) = \begin{cases} \text{accept} & \text{if } D \text{ does not accept } \llcorner D \lrcorner, \\ \text{reject} & \text{else if } D \text{ does accept } \llcorner D \lrcorner. \end{cases}$$

No matter what D does, it is forced to do the opposite. And this is a contradiction. The only possible resolution is that neither the TM D, nor the TM A can exist in the first place. ∎

Theorem 5.9 The language HALT defined in Eq. (13.1) is also undecidable.

Given that ACCEPT is undecidable, this claim should be hardly surprising. After all, HALT seems to be a more difficult language than ACCEPT. The proof of Theorem 5.9 is, in fact, based on this intuition and called a *reduction*. We show that a hypothetical TM (algorithm) that decides HALT could also be used to decide ACCEPT. This, however, leads to a contradiction, because we already know that ACCEPT is undecidable. proof by reduction

Proof of Theorem 5.9. Suppose that a TM H existed that decides HALT. Then, we can use H to construct a TM A that decides ACCEPT as well. This new TM processes inputs of the form $\llcorner(M, w)\lrcorner$ by executing (at most) two steps:

1) run TM H on input $\llcorner(M,w)\lrcorner$ to decide whether M halts on input w.

2.a) if it does not halt, output *reject* (if $M(w)$ does never halt, it will also never accept);

2.b) else if $M(w)$ does halt at some point, we can use a universal TM to simulate TM M on input w. We output *accept* if $U(\llcorner(M,w)\lrcorner) = M(w) =$ accept and *reject* if $U(\llcorner(M,w)\lrcorner) = M(w) =$ reject.

By construction, such a combination of TM H and a universal TM U would correctly decide the language Accept. This, however, is impossible. And, because we know that the universal TM U has to work, the problem must be the other TM H. The only solution is that it is impossible to decide the halting problem. ■

5.3.3 Two uncomputable languages

uncomputable = not semidecidable

We say that a language is *uncomputable* if it is not even semidecidable. This is the hardest class of languages (decision problems) conceivable. The complements of the two undecidable languages we have considered – Accept and Halt – are exemplary for this 'nastiest' class of decision problems.

Theorem 5.10 The language co-Accept introduced in Eq. (5.4) is *uncomputable*.

Although this theorem looks even stronger than the mathematical results from the last section, it turns out to be almost an immediate consequence of them. The only missing ingredient is the following logical implication.

Lemma 5.11 Suppose that a language $A \subseteq \Sigma^*$ and its complement co-$A = \bar{A}$ are both semidecidable. Then, A must, in fact, be decidable.

We leave the proof as a simple exercise.

Proof of Theorem 5.10. By contradiction. Recall that we have already shown that Accept is semidecidable (Lemma 5.5). Suppose now, that co-Accept were semidecidable as well. Then, Lemma 5.11 would imply that Accept must actually be decidable. This, however, is in direct contradiction to Theorem 5.8. Hence, co-Accept cannot be semidecidable to begin with. ■

Finally, let us turn our attention to the complement of the halting problem. By now, it should not come as a surprise that this language is uncomputable as well.

Theorem 5.12 The language co-Halt introduced in Eq. (5.4) is *uncomputable*.

The proof is almost identical to the proof of Theorem 5.10 and we leave it as a (very easy) exercise. Theorem 5.12 was originally proved by Turing, but we have presented a slightly more modern derivation.

5.4 Interpretations and implications

It is worthwhile to take some time to reflect about the mathematical proof behind Theorem 5.8. The proof is based on contradiction, or, more precisely a paradox. Here is a famous and instructive example:

liar's paradox

I am never telling the truth (liar's paradox).

The liar's paradox is based on *self-reference* and *negation*. A sufficiently powerful entity – in this case, a person – makes a yes/no statement about themselves and subsequently negates it. This necessarily leads to a contradiction that cannot be resolved.

Theorem 5.8 followed from a similar type of argument. We forced a powerful entity – now a TM – to make a yes/no statement about (a binary encoding of) itself and negate it. As in the liar's paradox, this leads to a contradiction that cannot be resolved. And, in turn, languages like ACCEPT can not be decided in general. With one statement of this type, we had a foot in the door. It allowed us to establish a variety of similar and even stronger statements by cleverly combining a single paradox with logically sound arguments ('if this were possible, then that has to be possible as well').

It is worthwhile to emphasize that these results state that there is no general method (or algorithm or TM) that can determine whether algorithms (TMs) halt. But the plural is essential here. Fortunately, it is often still possible to unambiguously decide the halting behavior of a single algorithm (or TM M). This is what formal proofs of correctness are all about.

Conceptually similar results have been discovered by Kurt Gödel in the 1930s. Roughly speaking, Gödel's first incompleteness theorem states that no consistent system of axioms combinable by an effective procedure (i.e. an algorithm) is capable of rigorously proving all truths about the arithmetic of natural numbers. This statement actually predates the discovery of Theorem 5.12, which is due to Turing, but looks closely related. In fact, it is possible to deduce Gödel's incompleteness theorem from Theorem 5.9 by using the fact that Turing numbers allow to convert statements about TMs into statements about natural numbers and algorithms.

Gödel's incompleteness theorem

It is no coincidence that both Gödel and Turing (as well as other scientists, like Alonzo Church) thought very hard about ultimate limits of formal logic. In 1928, David Hilbert – one of the most influential mathematicians of all times – and Wilhelm Ackermann posed the so-called *Entscheidungsproblem*: find an algorithm to determine whether a given sentence of predicate logic is *valid*, i.e. whether it is true regardless of the specific objects and relationships being reasoned about. Gödel's incompleteness theorem indicates that a general solution to the Entscheidungsproblem may be impossible, Turing's result about the language HALT (and an independent result by Alonzo Church) actually proves it. In fact, it was the Entscheidungsproblem that motivated Turing to

Entscheidungsproblem

develop TMs in the first place – as a formal description of general algorithms, which hadn't been properly defined back then.

And these negative results have had profound implications on the philosophy of mathematics. Before Gödel, Church and Turing published their results, there had been widespread consensus that mathematical statements are either true or false. And diligence, creativity and mathematical rigor are enough to find out which one it is. But the halting problem and its variants show that this dualistic view is false: some mathematical statements are impossible to prove right or wrong in the first place.

6. Time-bounded computations

Date: 16 November 2023

Agenda

In the first part of this lecture, we have been mainly concerned with *computability*: can a certain decision problem (language) be computed at all? Today, we begin our discussion of *computational complexity*. We focus on decision problems that are computable (decidable) and ask: how expensive is the actual computation going to be?

As a motivation, we consider two related arithmetic problems – *multiplication* and *factorization* – where the algorithmic cost deviates substantially. One of these problems appears to be computationally much cheaper than the other. This discrepancy becomes particularly pronounced if we consider (very) large inputs. In fact, actual running times of algorithms (Turing machines) can be very complex expressions. But a rough simplification – the so-called big-O notation – will allow us to isolate the most important contributions.

Having identified *running time* as an important algorithmic resource, we move on to define *time complexity classes* – families of languages (decision problems) that can be decided in comparable running time. The problem class **P** contains all decision problems that can be solved in a running time that scales polynomially in input size. It contains all computational problems that are 'easy' for computers. Concrete examples are basic arithmetic operations, like addition, subtraction, multiplication and division, as well as basic logical operations. But other, seemingly more involved problems also belong to this class.

Agenda:

1. Motivation: multiplication vs. factorization
2. Big-O notation
3. Time complexity
4. **P** and **EXP**
5. Examples

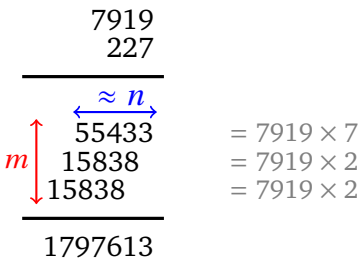


Figure 6.1 *Illustration of the gradeschool algorithm for multiplication:* $N = 7919$ is represented by $\lfloor \log_{10}(N) \rfloor + 1 = n = 4$ digits, while $M = 227$ is represented by $\lfloor \log_{10}(M) \rfloor + 1 = m = 3$ digits. The total number of elementary operations (single-digit multiplication and addition with carry-on) is approximately $2nm = 24$.

6.1 Motivation: multiplication vs. factorization

6.1.1 Multiplication

multiplication

Let $N, M \in \mathbb{N}$ be two (large) natural numbers comprised of $n = \lfloor \log_{10}(N) \rfloor + 1$ and $m = \lfloor \log_{10}(M) \rfloor + 1$ digits, respectively. Moreover, let us assume that $N \geq M$ (which also implies $n \geq m$). What is the computational cost of computing the *product*

$$P = N \times M?$$

Note that this question is rather vague. After all, there are different ways of computing such products. One of the simplest ways is *repeated addition:*

$$P = N \times M = \underbrace{N + \cdots + N}_{M \text{ times}}.$$

In words: we decompose our multiplication into a sequence of M additions. This is a lot of additions. And even if individual additions are cheap (which they are), this approach becomes really expensive once M becomes really big.

gradeschool multiplication

All of us know another way to compute products that scales much more favorably. It's the *gradeschool algorithm* we learned in middle school, see Figure 6.1 for an illustration. The trick is to decompose the big multiplication into a sequence of single-digit multiplications which are easy to look up (or memorize). We need roughly $n = \lfloor \log_{10}(N) \rfloor + 1$ elementary multiplications (with carry-over) to compute the product in each row. And, there are exactly $m = \lfloor \log_2(M) \rfloor + 1$ rows to compute and subsequently add up. The total cost is therefore dominated by

$$t_{\text{multiplication}} \approx nm \leq n^2, \tag{6.1}$$

where we have ignored constant pre-factors and subleading terms. The second relation follows from our assumption that $N \geq M$ (hence $n \geq m$). It highlights a quadratic scaling in the number of elementary operations (running time).

Note that the same approach would also work for binary representations of N and M. The total cost would be slightly higher – we need about $\log_2(10) \approx 3.3$-times more bits than digits to represent a number – but the resulting algorithm would be easier to implement on a computer and/or Turing machine (TM). Also, the main message is not affected by such a transformation: the number of arithmetic operations only depends on logarithms $n = \log(N)$, $m = \log(M) \leq n$ of the actual numbers involved. And, by definition, both n and m are *exponentially smaller* than N, M ($\exp(\log(x)) = \log(\exp(x)) = x$ for all $x \geq 0$). This underlines that computers (and TMs) are very good at computing products.

Example 6.1 (multiplication for two very large numbers). Fix $N = M = 2.4 \times 10^{67}$ (the number of atoms in the milky way galaxy). Then, we can use the gradeschool algorithm to compute the square $N^2 = N \times N = 5.76 \times 10^{134}$ by computing approximately $(\log_2(N))^2 \leq 224^2 = 50\,176$ elementary steps. This is completely feasible, modern hardware computes it in less than a second. ∎

It is worthwhile to point out that the gradeschool algorithm is not the fastest multiplication algorithm we know. An asymptotic running time proportional to $n \log(n)$ is possible by repeatedly applying the Fast Fourier transform. This also highlights that running time very much depends on the algorithm that is used to solve a certain problem.

6.1.2 Integer factorization

Integer factorization is the decomposition of a natural number N into a product of smaller integers. In a sense, this is the reverse operation to multiplication, e.g. factorization

$$1797613 = 7919 \times 227, \tag{6.2}$$

because 7919 and 227 are both prime numbers and cannot be factorized further. One of the most straightforward approaches for integer factorization, which is also taught in middle school, is *trial division* and was first described by Fibonacci (1202 AD): systematically test whether N is divisible by any smaller number. Start with 2 and see if $N/2$ is a natural number (no remainder). If this is the case, we have found our first factor: $N = 2 \times (N/2)$. Subsequently, we can focus our attention on further factoring $N/2$. If instead $N/2$ is not a natural number, 2 cannot divide N, and we move on to try division by 3 instead. We keep on repeating this procedure until we run out of possible candidate numbers. trial division

Trial division works relatively well if N is a product of many small prime numbers. For instance,

$$48 = 2 \times 24 = 2 \times 2 \times 12 = 2 \times 2 \times 2 \times 6 = 2 \times 2 \times 2 \times 2 \times 3 = 2^4 \times 3.$$

But, it can also become very labor-intensive if this is not the case. The worst case occurs if N is the product of two primes of comparable size. Let us take

Eq. (6.2) as a concrete example. In order to finally identify the smaller factor – 227 – we need to do a lot of trial divisions: 226 if we naively tried every possible number and 48 if we restricted our attention to prime numbers. The worst case actually occurs when N is the square of a prime $P = \sqrt{N}$. The naive method then requires a total of $\sqrt{N}$ trial divisions – each with a computational cost that scales roughly with $\log_{10}(N)^2$ (long division). Let $n = \lfloor \log_{10}(N) \rfloor + 1$ denote the number of digits of N. Then, the worst-case cost of naive trial division is

$$\tilde{t}_{\text{trial-division}} \approx \log_{10}(N)^2 \sqrt{N} \approx n^2 10^{n/2}, \tag{6.3}$$

where n is the number of digits contained in N. Restricting our attention to prime numbers only improves this scaling to

$$t_{\text{trial-division}} \approx \log_{10}(N)^2 \frac{\sqrt{N}}{\log_e(\sqrt{N})} \approx n 10^{n/2},$$

because the total count of prime numbers in the interval $[1, \sqrt{N}]$ is proportional to $\sqrt{N}/\log_e(\sqrt{N}) = 2\sqrt{N}/\log_e(N)$. This worst-case running time scales exponentially in the number of digits n. If we were to redo the same procedure in binary, we would also obtain an exponential scaling in the length of the input bitstring $|\llcorner N \lrcorner| = \lfloor \log_2(N) \rfloor + 1 \approx 3.3n$.

Example 6.2 (trial division for one very large number). Let M be a prime number in the ballpark of 2.4×10^{67} (the number of atoms in the milky way galaxy) and set $N = M \times M = M^2$. Then, we can use trial division to factor N into a product of prime numbers. This, however, would take astronomically many resources. The number of digits of M is (approximately) $n = 135$ which implies $t_{\text{trial-division}} \approx n 10^{n/2} \approx 4.3 \times 10^{69}$. This scaling estimate is itself larger than the number of atoms in the milky way galaxy! No computing machine existing (or even conceivable) can hope to handle this in the remaining lifetime of the universe (100 trillion years are approximately 3.1×10^{21} seconds – this number is 48 orders of magnitude (sic!) smaller than $t_{\text{trial-division}}$). ▪

A comparison between Eq. (6.2) and Eq. (6.1) seems to suggest that factorization can be exponentially more costly than multiplication. This, however, could simply be an indicator that trial division is a bad algorithm. But, here is the thing. Even the best factorization algorithms we know today ultimately still scale exponentially in the size of the input. Generations of algorithm designers have failed to come up with a solution whose (worst-case) running time scales polynomially in input size.

every traditional factorization algorithm has very bad scaling

And this may actually be a good thing. Multiplication and factorization are a pair of closely related problems that are certainly decidable. But the actual cost of deciding them is highly unbalanced. As the number of digits (or bits) becomes largish, one of them remains cheap, while the other one can be prohibitively expensive, even for the best-known algorithms running on the largest supercomputers of today. This discrepancy is actually used in cryptographic primitives, such as RSA (Rivest-Shamir-Adleman) encryption.

factorization as cryptographic primitive

There, the key idea is that users create a public key based on two large prime numbers (and an auxiliary value) that they publish. The actual prime numbers are kept secret, though. This ensures that anybody can encrypt messages, but they can only be decoded by someone who knows the prime numbers. The latter part is provided by the practical difficulty of factoring the product of two large prime numbers.

Although somewhat slow, RSA encryption is still widely used today. But, it may only be (computationally) secure against eavesdroppers that are restricted to conventional hardware. In 1991, Peter Shor developed an algorithm that can solve the decision-version of the factoring problem

$$\textsc{Factoring} = \{\llcorner (N,k) \lrcorner : \ N,k \in \mathbb{N} \text{ and } N \text{ has a factor less than } k\} \,. \quad (6.4)$$

in a running time that is (roughly) cubic in input size: $t_{\text{Shor}} \approx n^3$, where $n = \lfloor \log_2(N) \rfloor + 1$ is the bitlength required to store N. This is exponentially faster than the best known conventional running time. But, there is also a catch: Shor's algorithm can only be executed on a quantum computer. And the quantum computers we have built to date are far too small and far too noisy to actually challenge RSA encryption.

Exercise 6.3 Solve the factoring problem – for the special case where there are only two prime factors – by using a TM that decides the language FACTORING introduced in Eq. (6.4) as a subroutine. Show that the overall running time is dominated by the cost of deciding membership in FACTORING.
Hint: use binary search.

6.2 Big-O notation

By now, we have seen several examples of mathematical expressions that can become rather complicated and may even obscure what's really going on. Concrete examples include (i) the number of distinct pairwise distances in a traveling salesperson (TSP) instance (Lecture 1), (ii) the actual length of the bit encoding of a tuple (Lecture 5) and, (iii) the actual number of operations required to do trial division (Section 6.1.2 above).

Also, substantial deviations in running time – e.g. quadratic vs. exponential scaling – only really manifest themselves if we consider sufficiently large input sizes n. And for $n \gg 1$, these expressions are typically dominated by a single contribution that is most important. The *asymptotic notation* or *big-O notation* captures only this asymptotically leading term while also disregarding numerical scaling coefficients. This is best illustrated by a concrete example. Recall from Lecture 1 (and Exercise Sheet I) that a TSP instance involving n cities is fully specified by $T = n(n-1)/2$ pairwise distances. Asymptotically, this count becomes

big-O notation

$$T = \frac{n(n-1)}{2} = \frac{1}{2}n^2 - \frac{1}{2}n = O\left(n^2\right), \quad (6.5)$$

because n^2 grows quicker than n and we also disregard the scaling factor $1/2$. The right hand side (r.h.s.) of Eq. (6.5) succinctly tells us that a full description of TSP scales quadratically in the number of cities n. Another instructive example is the length of bit encodings of an alphabet Σ comprised of $|\Sigma| = N$ symbols:

$$n(N) = \lfloor \log_2(N) \rfloor + 1.$$

This expression has discontinuous jumps for certain alphabet sizes: $n(0) = n(1) = 1$, $n(2) = n(3) = 2$, $n(4) = \cdots = n(7) = 3$, and so on. This expression only starts to look nice if we choose N large enough, because then discontinuous jumps happen less frequently. Demanding $N \geq 2$ (or equivalently: $\log_2(N) \geq 1$) already ensures

$$n(N) = \lfloor \log_2(N) \rfloor + 1 \leq \log_2(N) + 1 \leq 2 \log_2(N) = O\left(\log_2(N)\right),$$

which is much easier to parse. Here is a formal definition of the big-O notation that takes into account that we ignore scaling factors and only consider inputs that are sufficiently large.

Definition 6.4 (big-O notation). Let $f, g : \mathbb{N} \to \mathbb{R}_+$ be two functions we wish to compare. We say that $f(n) = O\left(g(n)\right)$ if there is a constant $c > 0$ and a natural number $n_0 \in \mathbb{N}$ such that, for all $n \geq n_0$, $c \times g(n)$ is at least as large as $f(n)$:

$$f(n) = O\left(g(n)\right) \quad \text{if } f(n) \leq c \times g(n) \text{ for all } n \geq n_0.$$

This definition captures the notion that $g(n)$ grows faster than $f(n)$, provided that we compare both functions for large enough values n.

Example 6.5 $f(n) = (3n+2)^2 = 9n^2 + 12n + 4$ and $g(n) = n^2$. It is easy to check that $g(n) \leq f(n)$ for all $n \in \mathbb{N}$. A bit more work, moreover, yields $f(n) \leq 12 g(n)$ whenever $n > 4$. In other words,

$$9n^2 + 12n + 4 = f(n) = O\left(g(n)\right) = O\left(n^2\right).$$

and we refer to Fig. 6.2 for a visualization. This showcases the value of the big-O notation. It allows us to extract the essential growth behavior of complicated and/or annoying functions. ∎

Finally, it is worthwhile to point out that the big-O notation plays nicely with logarithms. In principle, the logarithm function only makes sense if we also specify a base. In particular, $\log_2(\cdot) \neq \log_e(\cdot) = \ln(\cdot) \neq \log_{10}(\cdot)$. But, changing the value of the base only changes the logarithm by a constant factor: for $b, c > 0$

big-O scaling of logarithms

$$\log_b(n) = \frac{\log_c(n)}{\log_c(b)} \quad \text{for all } n > 0.$$

And $1/\log_c(b)$ is constant for all reasonable changes of base, i.e. where both b and c are smallish constants (e.g. $2, \mathrm{e}, 10 \leq 10$). This, in turn, ensures

$$\log_b(n) = \frac{1}{\log_c(b)} \log_c(n) = O\left(\log_c(n)\right) \tag{6.6}$$

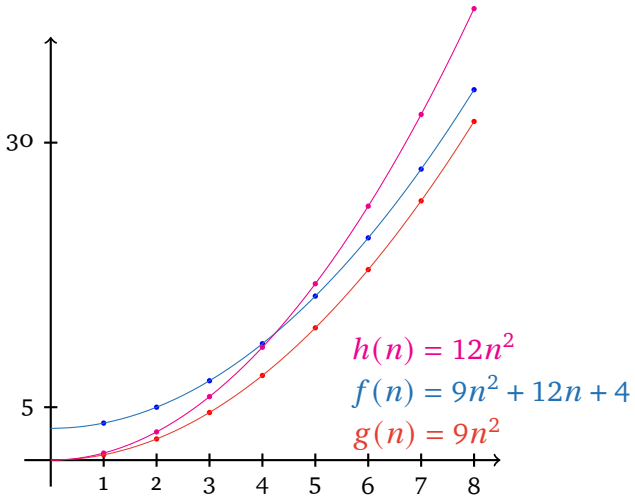


Figure 6.2 *Big-O analysis of a polynomial:* the function $f(n) = 9n^2 + 12n + 4$ (blue) is always larger than the simpler function $g(n) = n^2$ (red). But, once $n > 4$, $f(n)$ is actually smaller than $h(n) = 12n^2$ (magenta). This implies both $f(n) = \Omega(g(n)) = \Omega(n^2)$ and $f(n) = O(h(n)) = O(n^2)$ and we conclude $f(n) = \Theta(n^2)$. In fact, we can visually see that the growth behavior of all three functions is already quite similar for $n \geq 7$.

and the converse relation is true as well:

$$\log_c(n) = \log_c(b) \log_b(n) = O\left(\log_b(n)\right).$$

So, whenever we perform an asymptotic analysis, specifying bases of the logarithms involved is no longer necessary. This insight also justifies our lax treatment of logarithm bases in Section 11.1. A similar invariance property is also true for exponential functions. For instance,

big-O scaling of exponentials

$$10^n = \left(2^{\log_2(10)}\right)^n = 2^{\log_2(10)n} \approx 2^{3.3 \times n} = 2^{O(n)},$$

and this relation readily extends to numbers other than 10 or 2. This should not come as a surprise. Exponentiation and taking logarithms are closely related after all. (In fact, you can use this relation to prove the change of base formula (6.6) in the first place.)

6.3 Time complexity

We have now assembled all necessary pieces to reason about *computational complexity*, i.e. the actual cost of solving computational problems. Unless otherwise specified, we will focus on decision problems or languages that are *decidable*. That is, we will only consider problems where we know that a Turing machine (TM) is capable of producing a solution. Note that this is roughly

equivalent to demanding that an actual computer is capable of solving the problem in question. There are different ways of quantifying the cost of a computation. Arguably the most intuitive one is *running time*: how long does it take a TM (or computer) to solve certain tasks? running time

The concept of a Turing machine as our computational model is very helpful here. It allows us to equate running time with the number of steps a TM needs to either accept or reject a given input. This running time can be very complicated and typically depends on several parameters. In particular, longer inputs typically require more computational steps. For simplicity and clearness, we compute running times only in terms of input size. This is a crude simplification, because we ignore problem-specific structure, but it often allows us to get an intuitive high-level view on how these problems behave. Moreover, we also adopt a pessimistic viewpoint and focus on *worst case* inputs.

Definition 6.6 (running time or time complexity). Let M be a Turing machine that halts on all inputs. The *running time* or *time complexity* of M is the function $t : \mathbb{N} \to \mathbb{N}$ that counts the maximum number of steps that M uses on any input length n.

Definition 6.7 (time complexity classes). Let $f : \mathbb{N} \to \mathbb{R}_+$ be a function. A language A is contained in the *time complexity class* $\mathbf{DTIME}(f(n))$ if there exists a Turing machine M that decides A in a running time that satisfies $t(n) = O(f(n))$ for all input sizes $n \in \mathbb{N}$.

Throughout the course of this lecture, we have already seen a fair share of languages. It is instructive to point out the time complexity class they actually belong to.

Example 6.8 (palindrome). The language $\textsc{Palindrome} = \left\{x \in \{0,1\}^* : \ x^{\mathrm{R}} = x\right\} \subset \{0,1\}^*$ is contained in the complexity class $\mathbf{DTIME}(n^2)$. This tells us that the running time of checking whether a bit string equals its own reverse can at most scale quadratically in the size of the input. This follows from analyzing the running time for the specific TM we constructed in Lecture 3. We denote this observation by writing $\textsc{Palindrome} \in \mathbf{DTIME}(n^2)$. ▪

Example 6.9 (regular languages). Every regular language $A \subseteq \{0,1\}^*$ obeys $A \in \mathbf{DTIME}(n)$. This follows from the fact that we can simulate finite state automata with a TM that executes exactly n steps on inputs of length n (ignore the tape and process input bits one by one). This is, in particular, true for computing parity: $\textsc{Parity} = \left\{x \in \{0,1\}^* : \text{ the last bit is a } 1\right\} \in \mathbf{DTIME}(n)$. ▪

Exercise 6.10 (SAME; challenging). It is easy to come up with a TM ('algorithm') that also decides the language $\mathrm{SAME} = \left\{1^k 0^k : \ k \in \mathbb{N}\right\} \subset \{0,1\}^*$ in quadratic time ($\textsc{Same} \in \mathbf{DTIME}\left(n^2\right)$). Devise a better TM ('algorithm') that actually achieves the same task in only $O\left(n \log(n)\right)$ steps, where n is the length of the input bitstring. This improvement implies $\textsc{Same} \in \mathbf{DTIME}(n \log(n))$.

Example 6.11 (quantum algorithms). Quantum computers are a completely different type of hardware that allows us to execute algorithms which conventional

hardware do not support at all. Concepts like running time do carry over to quantum algorithms. In particular, it makes sense (with some caveats that we sweep under the rug for now) to say that a quantum computer requires $f(n)$ computational steps to decide a given input string of length n. E.g. Shor's algorithm solves the factorization problem in $f(n) = O(n^3)$ steps.

However, quantum computers can in principle be simulated by conventional software (or TMs), but there is an exponential overhead in input size. Simulating Shor's algorithm on a conventional TM therefore ensures $\textsc{Factoring} \in \mathbf{DTIME}\left(2^{O(n^3)}\right)$. This conclusion is interesting, but weaker than analyzing direct solution algorithms. Trial search, for instance, already implies $\textsc{Factoring} \in \mathbf{DTIME}\left(2^{O(n)}\right)$.

∎

6.4 The time complexity classes P and EXP

We are now ready to introduce our first, and probably most important, computational complexity class:

the complexity class **P**

$$\mathbf{P} = \bigcup_{k\geq 1} \mathbf{DTIME}\left(n^k\right) = \mathbf{DTIME}\left(n\right) \cup \mathbf{DTIME}\left(n^2\right) \cup \cdots$$

Informally speaking, **P** contains all (decidable) problems that can be solved in polynomial running time. This will be our proxy for *efficiently solvable*. Looking at this definition, it seems questionable whether such a crude union of *all* polynomial-time classes makes sense at all. For instance, should we call a TM ('algorithm') whose running time scales with $O\left(n^{100}\right)$ an efficient algorithm? This high-level classification only really starts to make sense if we compare **P** to its much larger relative:

the complexity class **EXP**

$$\mathbf{EXP} = \bigcup_{k\geq 1} \mathbf{DTIME}\left(2^{n^k}\right) = \mathbf{DTIME}\left(2^n\right) \cup \mathbf{DTIME}\left(2^{n^2}\right) \cup \cdots$$

Informally speaking, **EXP** contains all (decidable) problems that can be solved in exponential running time. A running time of $O\left(n^{100}\right)$ is certainly terrible, but it pales in comparison to exponential running times, like $2^{n^{100}}$. It is easy to check that

$$\mathbf{P} \subseteq \mathbf{EXP},$$

because the running time of every polynomial-time TM is always upper-bounded by some exponential function. It also seems reasonable to assume that **P** is strictly smaller than **EXP**, i.e. there are problems that require exponentially long running time in the worst case. The factoring problem from above and the TSP problem come to mind in this context.

It is actually possible to turn this intuition into a rigorous statement. This is the content of the *time hierarchy theorem* which applies to running time scalings $f : \mathbb{N} \to \mathbb{R}_+$ that are 'reasonable' in the sense that for $n \in \mathbb{N}$, $f(n)$ can be computed in $O(f(n))$-running time. Such functions are called *time-honest*.

Theorem 6.12 (time hierarchy theorem, informal). Let $f : \mathbb{N} \to \mathbb{R}_+$ be a time-honest function. Then, there exists a language A that can be decided in running time $O(f(n))$, but not in running time $O(f(n)/\log_2(n))$.

time hierarchy theorem

Alas, we don't have time for a complete formal proof which is conceptually similar to the proof that the halting problem is not decidable. The key idea is to use the assumption that $f(n)$ is time-honest to reason about the language that contains all TMs M which accept an input x after (at most) $f(|x|)$ computing steps. This language is rich enough to use self-reflection and negation to construct a contradiction that arises if membership in this class could be determined in a running time that is somewhat smaller than $O(f(|x|))$. Note also that we will derive hierarchy theorems for other complexity classes later in the course. For now, let us instead focus on the implications of Theorem 6.12. Applying it to $f(n) = n^3$, for instance, highlights that there are languages $A \in \mathbf{DTIME}(n^3)$, but $A \notin \mathbf{DTIME}(n^3/\log_2(n))$. We can then use $O(n^2) = O(n^3/\log_2(N))$ to infer $A \notin \mathbf{DTIME}(n^2)$. In words: there are languages that can be decided in cubic running time, but not in quadratic running time . The following corollary follows from a similar chain of arguments.

Corollary 6.13 (P $\neq$ EXP). There are languages A which can be decided in exponential running time, but not in polynomial running time: $A \in \mathbf{EXP}$, but $A \notin \mathbf{P}$.

P $\neq$ EXP

Proof. Apply the time hierarchy theorem to the function $f(n) = 2^n$ (which is time-honest). This ensures that there are languages A such that $A \in \mathbf{DTIME}(2^n) \subseteq \mathbf{EXP}$, but $A \notin \mathbf{DTIME}(2^n/\log_2(n))$. Next, note that for any $k \in \mathbb{N}$, $O(n^k) = O(2^n/\log_2(n))$ (exponential growth eventually outraces any polynomial growth even when suppressed by a logarithm). So, $A \notin \mathbf{DTIME}(2^n/\log_2(n))$ also implies $A \notin \mathbf{DTIME}(n^k)$ for any $k \in \mathbb{N}$. This, however, is equivalent to $A \notin \mathbf{P}$. ■

Let us conclude this section with emphasizing another conceptual advantage of summary classes like **P** and **EXP**. Changes in the actual computational model only lead to running time overheads that are (at most) polynomial in running time. (There is one possible exception that we discuss in Sub. 6.5.4 below). E.g. converting a multi-tape Turing machine into a single-tape Turing machine will in general produce a quadratic overhead. And the complexity classes **P** and **EXP** are closed under such transformations. Even better, this invariance persists if we switch from TMs to actual algorithm descriptions (in pseudo-code) and vice versa.

the computational model doesn't matter

Fact 6.14 When talking about large problem classes, like **P** and **EXP**, we can typically ignore implementation-specific details of the actual algorithm. ▪

Needless to say, it is good scientific practice to carefully double-check such invariance properties in every concrete complexity problem you wish to tackle.

6.5 Example problems

6.5.1 Elementary algebraic operations

The decision problem versions of elementary algebraic computations are contained in **P**. For multiplication, one possible formulation is

$$\textsc{Multiply} = \{\llcorner(a,b,c)\lrcorner : \ a,b,c \in \mathbb{N} \text{ and } c = a \times b\} \subset \{0,1\}^* .$$

The running time for deciding membership scales (at most) quadratically in the length of the encoded tuple. This, in turn, asserts

$$\textsc{Multiply} \in \mathbf{DTIME}\left(n^2\right) \subset \mathbf{P}.$$

We can draw similar conclusions for language versions of the other elementary arithmetic operations:

$$\begin{aligned}
\textsc{Add} &= \{\llcorner(a,b,c)\lrcorner : \ a,b,c \in \mathbb{N} \text{ and } c = a + b\} \subset \{0,1\}^* , \\
\textsc{Subtract} &= \{\llcorner(a,b,c)\lrcorner : \ a,b,c \in \mathbb{N} \text{ and } c = a - b\} \subset \{0,1\}^* , \\
\textsc{Divide} &= \{\llcorner(a,b,c)\lrcorner : \ a,b,c \in \mathbb{N} \text{ and } c = a/b\} \subset \{0,1\}^* ,
\end{aligned}$$

are all also contained in **P**. We can use bit-wise addition/subtraction (with carry-over) to solve Add and Subtract in linear running time $O(n)$. Long division, like we learned it in middle school, can be converted into an $O(n^2)$-algorithm. And the overhead for implementing all these algorithms on a TM is at most polynomial.

6.5.2 Elementary logical operations

In formal logic, we typically work with binary variables $x_0, \ldots, x_{n-1} \in \{0,1\}$ that indicate whether something is true ($x_i = 1$) or false ($x_i = 0$). A *Boolean formula* combines variables with logical operations like $\wedge$ ('and'), $\vee$ ('or') and $\neg$ ('negation'), as well as parentheses that tell us in which order we execute these operations.

Example 6.15 (logical equality). The following Boolean formula involves only two variables x_0, x_1:

$$\varphi(x_0,x_1) = (\neg x_0 \vee x_1) \wedge (x_0 \vee \neg x_1) \in \{0,1\} . \tag{6.7}$$

It evaluates to 1 ('true') if and only if $x_0 = x_1$. ∎

As a rule of thumb, $\wedge$ behaves similarly to multiplication, while $\vee$ is reminiscent of addition. So, to process Boolean formulas as efficiently as possible, it is often useful to bring it into *conjunctive normal form* (CNF), an AND of ORs with possible negations (think: a product of sums). The formula $\varphi(x_0,x_1)$ defined in Eq. (8.3), for instance, is in CNF. It is worthwhile to recall that every possible Boolean function can be converted into an equivalent formula that is in CNF. For instance, suppose that n is a multiple of 3. Then,

conjunctive normal form

$$\varphi\left(x_0,\ldots,x_{n-1}\right) = (x_0 \vee x_1 \vee x_2) \wedge \cdots \wedge (x_{n-3} \vee x_{n-2} \vee x_{n-1})$$

is a Boolean formula in n variables in CNF. The bracketed expressions are called *clauses*. Each of the clauses contains exactly $l = 3$ variables and there are $m = n/3$ of them. Computing $\varphi(x_0, \ldots, x_{n-1})$ for a fixed input bitstring is not difficult. The truth value of each clause can certainly be evaluated in running time $O(3n) = O(n)$, because there are exactly $l = 3$ variables per clause and it may take up to $O(n)$ TM-steps to look up the relevant truth values. We need to do this subroutine m times – once for each clause – and form the logical AND of all resulting truth values. The total cost is roughly m times the cost of computing an individual clause.

clauses

This procedure generalizes to arbitrary Boolean formulas in CNF. Let n be the number of variables, m the number of clauses and l the maximum number of variables within one clause. Then, the running time of evaluating this Boolean function is (at most)

$$t_{\text{Boolean-function}} = O(mnl) = O\left((ml\log_2(n) + n)^2\right), \tag{6.8}$$

The last step in Eq. (6.8) is a very rough upper bound. But $ml\log(n) + n$ is roughly the number of bits that are required to specify both the Boolean formula $\varphi(\cdot)$ and an input $(x_0, \ldots, x_{n-1})$:

$$O(ml\log_2(n) + n) = O(|\llcorner\varphi\lrcorner| + |\llcorner x\lrcorner|) = O(|\llcorner(\varphi, x)\lrcorner|). \tag{6.9}$$

Comparing Eq. (6.8) and Eq. (6.9) asserts that the cost of computing a given Boolean formula in CNF form scales at most quadratically in the actual size of the input. Hence, the decision version of Boolean function evaluation

$$\textsc{BooleanEvaluation} = \left\{\llcorner(\varphi, x)\lrcorner \in \{0,1\}^* : \ \varphi \text{ is CNF and } \varphi(x) = 1\right\}$$

obeys

$$\textsc{BooleanEvaluation} \in \text{DTIME}(n^2) \subset \mathbf{P}.$$

This shows that evaluating Boolean functions is also cheap, as it should be. Computers are, after all, very good at performing logical operations.

6.5.3 Computing the determinant of a matrix

Finally, let us consider a computational problem from linear algebra, where it is somewhat surprising that efficient algorithms actually exist. The *determinant* of a symmetric $n \times n$ matrix X (a 2D array with n columns and n rows where each entry is a real valued number) is defined by the Leibnitz rule:

$$\det(X) = \sum_{\sigma \in S_n} \text{sgn}(\sigma) \prod_{i=1}^{n} x_{i,\sigma(i)}, \tag{6.10}$$

where $x_{ij} \in \mathbb{C}$ denotes the entry of X in the i-th row and j-th column and the sum ranges over all possible permutations σ of n elements. Finally, $\text{sgn}(\sigma) \in \{\pm 1\}$ is a permutation-specific signature that can be either $+1$ or -1. Eq. (6.10) looks scary, the sum alone ranges over $n! = n(n-1)\cdots n$ different permutations. This number explodes faster than exponential, i.e. $2^n = O(n!)$.

Nonetheless it is possible to actually compute $\det(X)$ in cubic time, i.e. the decision variant of determinant obeys $\det(X) \in \mathbf{DTIME}(n^3) \subset \mathbf{P}$. The trick is to exploit a very nice property of the determinant. It is invariant under orthogonal transformations, aka basis changes: $\det(X) = \det(OXO^T)$ for any O that obeys $OO^T = O^TO = \mathbb{I}$ (orthogonality). This allows us to first bring our matrix X into diagonal form $X = ODO^T$ and compute the determinant afterwards. This diagonalization only requires $O(n^3)$ operations and makes a huge difference. Computing Eq. (6.10) for diagonal matrices D is super-simple, because most matrix entries are zero: $\det(D) = \prod_{i=0}^{n-1} x_{ii}$. This extra step only requires $O(n)$ operations and pales in comparison to the $O(n^3)$-cost of diagonalizing the matrix in the first place.

6.5.4 Criticisms and extensions

It is worthwhile to spell out and discuss some possible criticisms of the definition of **P**. The following list of arguments is taken from [AB09].

1. *Worst-case running time is too strict:* in our definition of running time, we only consider algorithms that decide a language (answer a yes/no question) *exactly* and with bounded running time for every possible input. However, it is extremely rare that all possible input strings do actually arise in practice. Theoretical computer scientists are very aware of this drawback and have also studied *average-case complexity* analogues of time complexity classes, like **P** and **EXP**. This, however, would go beyond the scope of this introductory lecture.
2. *Decision problems are too limited:* Some computational problems cannot be straightforwardly expressed as decision problems. The original traveling salesperson problem – find the shortest route that visits a total of n cities exactly once – is one such example. Fortunately, the framework of decision problems turns out to be surprisingly expressive. In particular, it is possible to solve this problem by repeatedly answering the decision version of TSP – does a route exist that is shorter than k kilometers? – for different values of k. This incurs only a logarithmic overhead and is part of Exercise Sheet III.
3. *Other physically realizable models of computation:* our concept of running time ultimately depends on the computational model of a Turing machine. But, the vision of quantum computers and quantum algorithms has challenged this foundation. By now, theoretical quantum computer scientists have extended most complexity-theoretic concepts from TMs to quantum computers. This led to a rich and interesting theory about quantum complexity classes. In particular, there are well-defined generalizations of **P** – the complexity class **BQP** – and **NP** – the complexity class **QMA**. The class **BQP**, however, may be strictly larger than its conventional counterpart. Shor's algorithm, in particular, implies FACTORING $\in$ **BQP**, but we don't know whether FACTORING $\in$ **P** (because we don't know an efficient conventional algorithm). The larger expressivity of **BQP** is one

of the main reasons why we try so hard to actually build a fully-functional quantum computer. Alas, this fascinating topic is also beyond the scope of this introductory course.

7. The problem class NP

Date: 23 November 2023

Agenda

Last time we have introduced two our first two computational complexity classes. **EXP** is the class of decision problems that can be solved in a running time that scales exponentially in input size (on a Turing machine). **P**, in contrast, contains those that can be solved in polynomial running time. Today, we will introduce another complexity class that lies somewhere between these two extremes, see Fig. 7.1 for an illustration. The class **NP** contains all problems that may be difficult to solve, but where it is at least possible to efficiently check whether a proposed solution checks out. This sets the stage for the famous **P** vs. **NP** conjecture: is checking correctness of a solution easier than coming up with the solution in the first place? Or, is the difficulty of both tasks actually comparable? Despite decades of intensive research, the answer to this question remains unknown. **P** vs **NP** is one of the 7 *Millenium Prize Problems*.

Agenda:

1. Motivation: Factoring
2. The problem class **NP**
3. Examples
4. Origin story
5. Philosophical implications

7.1 Motivation: Factoring

Recall that *Factoring* is the task of decomposing a large number $N \in \mathbb{N}$ into a product of prime factors. If N is the product of two large prime numbers ($N = F_0 \times F_1$), the best known algorithm requires a running time that scales exponentially in the number of digits involved. This leads us to believe that factoring is a difficult problem. In fact, we actually build encryption protocols based on that belief. The most widespread language variant of factoring is

$$\textsc{Factoring} = \{\llcorner (N, k) \lrcorner : \ N, k \in \mathbb{N} \text{ and } N \text{ contains a factor} \leq k\} . \quad (7.1)$$

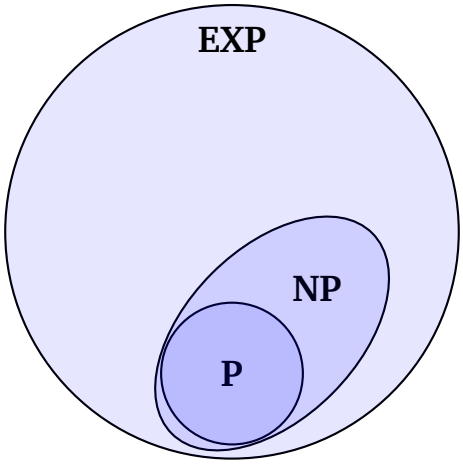


Figure 7.1 *Landscape of complexity classes (to be continued):* The class **EXP** is a large complexity class that encompasses all decision problems (languages) that can be decided in exponential running time (in input size). Much smaller by comparison, **P** only contains those that require polynomial running time. The problem class **NP** is in-between. It contains decision problems that may be hard to solve, but proposed solutions can be efficiently checked for correctness.

It does not solve factoring directly. But, when combined with binary search, it can be empowered to do just that. This will be the content of Problem 7.12 below. Clearly, FACTORING $\in$ **EXP** (why?), but it does have one redeeming quality: If you manage to obtain a solution, you can readily convince others that your solution is indeed correct. Suppose you have discovered $\llcorner (N, k) \lrcorner \in$ FACTORING for some N and some k and let us assume for simplicity that N is the product of only two primes F_0 and F_1. Then you can prove the correctness of your claim by revealing the actual factor you found, say F_0. Access to (a bit representation of) this number allows your peers to check

(i) N/F_0 is integer i.e. F_0 is indeed a factor;
(ii) $F_0 \leq k$, i.e. one of the factors is indeed at most k.

Crucially, sharing F_0 only requires distributing a total of $n = O(\log(F_0)) = O(\log(N))$ bits and both validity checks are *efficient* in terms of running time. Division is an arithmetic operation whose running time is $O\left(\log(N)^2\right)$ and comparing F_0 with k only requires $O(\log(F_0))$ operations. What is more, these consistency checks cannot be fooled. If N does not have a factor $F_0 \leq k$, then it is impossible to come up with a 'fake factor' that tricks your peers into believing $(N, k) \in$ FACTORING.

7.2 The problem class NP

Although factoring may be exponentially hard to solve in general, it is at least easy to check the correctness of a proposed solution. Being able to convince

others of the correctness of an argument is an important driving force for scientific and technological advancement. The problem class **NP** captures precisely these types of problems.

NP = efficiently verifiable

Definition 7.1 (NP, informal). The problem class **NP** contains all languages (decision problems) where membership can be efficiently verified.

This is a crisp and intuitive definition. But behind it, there is actually quite a lot to unpack. Let $A \subseteq \{0,1\}^*$ be a language. Then, Definition 7.1 actually demands two things:

completeness and soundness

- **completeness:** if $x \in A$, then there is a (pre-specified) verification procedure and a short certificate y that allows others to verify $x \in A$ in reasonable time ('all yes-instances are covered by the same type of argument');
- **soundness:** else if $x \notin A$ there cannot be a certificate y that fools others to wrongfully believe $x \in A$ ('no false positives').

Example 7.2 (FACTORING is complete and sound). The language FACTORING introduced in Eq. (11.1) is both complete and sound. It is complete, because for each $\llcorner(N,k)\lrcorner \in$ FACTORING, there exists a short certificate – the factor M that obeys $N/M \in \mathbb{N}$ and $N \leq k$. It is also sound, because it is impossible to 'fake' a number M with these properties if $\llcorner(N,k)\lrcorner \notin$ FACTORING. ∎

So far, we have introduced quite a bit of jargon, like 'short', 'verification procedure' and 'efficient'. The computational models introduced earlier in this course allow us to be more specific. Input x and certificate y can always be represented by bitstrings. The certificate y is *short* if its length scales (at most) polynomially in the length of the actual input. I.e. $|y| = \text{poly}(|x|)$ (here, $\text{poly}(n)$ denotes an arbitrary polynomial in n, e.g. n, n^2, n^3, etc.). This takes into account that larger inputs may require longer certificates. A (pre-specified) *verification procedure* is a fixed TM M (think algorithm) that can process inputs $\llcorner(x,y)\lrcorner$ of arbitrary length and either accepts or rejects them. We call such a verification *efficient*, if its running time scales (at most) polynomially in input size $|\llcorner(x,y)\lrcorner| = 2\,(|x| + |y| + 1) = \text{poly}(|x|)$, because $|y| = \text{poly}(|x|)$.

NP, formal definition

Definition 7.3 (NP, formal definition). A language $A \in \{0,1\}^*$ is in **NP** if there exist polynomials $p, q : \mathbb{N} \to \mathbb{N}$ and a TM M with the following two properites:

- **Completeness:** if $x \in A$, then there exists a *short certificate* $y \in \{0,1\}^*$ with $|y| = p(|x|)$ such that $M(x,y) =$ accept after (at most) $q(|x|)$ steps.
- **Soundness:** else if $x \notin A$, then $M(x,y) =$ reject for all possible certificates $y \in \{0,1\}^*$.

This definition spells out all mathematical details that are required to make rigorous statements about the intuition provided in Definition 7.1. The following inclusion properties, for instance, are now easy to verify and justify the illustration in Figure 7.1.

Proposition 7.4 (inclusion properties).

1. **P** $\subseteq$ **NP**, i.e. every problem that is easy to solve is also easy to verify.
2. **NP** $\subseteq$ **EXP**, i.e. problems in **NP** may be hard, but they are always solvable.

We leave the proofs as an exercise and conclude this section with emphasizing one of the most important open problems in computer science.

Computational Primitive (P vs. NP conjecture). **P** $\neq$ **NP**, i.e. checking the correctness of a solution is (sometimes) easier than comping up with the solution yourself.

P vs. **NP** conjecture

It is widely believed that **P** $\neq$ **NP**, but a rigorous proof still eludes us. The **P** vs. **NP** conjecture is actually one of only 7 Millenium Prize problems[1]. That means, solving it would award USD 1 000 000, as well as eternal fame.

7.3 Examples

Example 7.5 (Factoring). Given two numbers N, k, decide whether N has a factor smaller than k. The certificate is the factor M. The verifying TM checks (i) whether M divides N and (ii) $M \leq k$. ∎

Example 7.6 (Travelling salesperson). Given a table D of pairwise distances between n cities, decide whether there is a route that visits all cities exactly once and has total length at most k. The certificate is the route r, the verifier computes $\mathrm{km}(r)$ and checks $\mathrm{km}(r) \leq k$. ∎

Example 7.7 (Boolean satisfiability). Given a polynomial-sized Boolean formula $\varphi : \{0,1\}^n \to \{0,1\}$ in conjunctive normal form (CNF), decide whether there exists an input $x_0, \ldots, x_{n-1} \in \{0,1\}$ such that $\varphi(x_0, \ldots, x_{n-1}) = 1$. The certificate is one such bitstring $(y_0, \ldots, y_{n-1})$, the verifier computes $\varphi(y_0, \ldots, y_{n-1})$ and checks whether it is equal to 1. ∎

Example 7.8 (Graph isomorphism). A graph (V, E) is a mathematical object that combines a set of vertices (nodes) V with edges $e \in E$, i.e. connections between two vertices. Two graphs (V_1, E_1) and (V_2, E_2) are isomorphic if it is possible to convert one into the other by rearranging the vertices. Checking graph isomorphism is in **NP**. The certificate is a permutation of vertices that does the job. The verifier checks whether the newly permuted graph is equal to the other one. ∎

7.4 Origin story: non-deterministic Turing machines

In this lecture we have mainly focused on computing devices that are *deterministic*. The prime example is a deterministic finite automaton (DFA). We have also modelled the tape head of our TMs by a DFA which results in a TM that is also deterministic. But this is not the only choice. One can also define a

[1] See https://www.claymath.org/millennium-problems

variant of TMs, where the tape head is described by a *nondeterministic* finite state automaton (NFA). This results in a different computational model, called a *nondeterministic Turing machine*.

Before jumping into details, it is worthwhile to change our perspective about NFAs a bit. A DFA is a 5-tuple $M = \big(Q, \Sigma, \delta, q_0, q_{\text{accept}}\big)$, where $\delta : Q \times \Gamma \to Q$ is a deterministic transition function. That is, for each state $q \in Q$ and each symbol $x \in \Sigma$, there is exactly one valid state transition $\delta(q, x) = p$. For NFAs, this is not the case anymore. There can be more than one possible transition. This gives rise to nondeterministic choices on how to proceed. But, we can model them by assigning multiple deterministic transition functions $\delta_0, \ldots, \delta_{c-1}$ ($c \leq |\Sigma|$) to the finite state automaton. Nondeterminism manifests itself in the ability to select one of these transition functions at every step of the computation. And we accept a length-n input string if and only if at least one subselection of n transition functions $\delta_{i_0}, \ldots, \delta_{i_{n-1}}$ terminates in the accepting state q_{accept}.

Nondeterministic **T**uring **M**achines (NTM)

This concept readily extends to Turing machines. A non-deterministic Turing machine (NTM) is a TM $M = \big(Q, \Sigma, \Gamma, \{\delta_0, \ldots, \delta_{c-1}\}, q_0, q_{\text{accept}}, q_{\text{reject}}\big)$ with multiple deterministic transition functions $\delta_i : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$. At each step of the computation, we now have the freedom to choose one of these transition functions at will. And, we say that a NTM accepts input x if there is (at least) one sequence of these choices that would let M reach q_{accept} on input x.

NP abbreviates **N**ondeterministic **P**olynomial time

Definition 7.9 (nondeterministic polynomial time). The problem class **NP** contains all languages (decision problems) that can be decided by a nondeterministic Turing machine in polynomial running time. In fact, **NP** is an abbreviation for **N**ondeterministic **P**olynomial (running) time.

Theorem 7.10 Definition 7.9 and Definition 7.3 are equivalent.

Proof sketch. We need to show two directions.

- $\Rightarrow$ Suppose that a language A is decided by a NTM $\tilde{M}$ that executes at most $T = \text{poly}(n)$ steps. According to Definition 7.9, this implies that for every $x \in A$, there is a sequence of nondeterministic transition function choices that reach q_{accept} on input x. We can interpret this sequence of choices as a certificate for x. The string $y = \llcorner(y_0, \ldots, y_{T-1})\lrcorner$ with $y_k \in \{0, \ldots, c-1\}$ encodes which of the c transition functions we applied at each step to determine $x \in A$. This certificate has polynomial length $O\left(\log(c)T\right) = O(p(n))$. What is more, we can construct a deterministic TM that checks whether $\tilde{M}$ really enters the accepting state q_{accept} after executing this sequence of transitions on input x. Hence, $A \in$ **NP** according to Definition 7.3.
- $\Leftarrow$ Conversely, assume that A is a **NP**-language according to Definition 7.3. Then, we can construct a polynomial-time NTM that also decides A. The key idea is to use non-deterministic 0/1 choices to write down a

string y of length $|y| = \text{poly}(x)$. (Concretely, this is possible by using two additional transition functions: δ_0 writes down 0 and moves to the right, while δ_1 writes down 1 and moves to the right). Importantly, all possible strings with (the correct) polynomial length are possible. Then, we run the original DTM M on the enlarged input string xy. It accepts if and only if a verifier y exists. Moreover, the entire procedure runs in nondeterministic time $|y| + \text{runningtime}(M) = \text{poly}(|x|)$.

■

7.5 Philosophical implications of P vs. NP

On an abstract level **P** vs. **NP** is a question about the power of nondeterminism in Turing machines. Recall that this question is fully understood for finite state automata. We mentioned in Lecture 2 that a nondeterministic finite state automaton (NFA) can always be converted into a functionally equivalent deterministic finite state automaton (DFA). And, while the required number of states may grow exponentially, the actual running time remains the same. Both DFAs and NFAs have linear running time by construction. **P** vs. **NP** asks whether a similar equivalence is true for Turing machines as well.

But, there are more practical implications as well. Often, **NP** problems seem difficult, because they involve some sort of exhaustive search. Concrete examples are TSP (search over all routes to find the shortest one), factoring (search over all possible prime numbers to find a factor), Boolean satisfiability (search over all possible truth assignments to find one that is satisfiable). And for each of these problems, the 'fitness' (cost function) of each trial is easy to evaluate. **P** vs. **NP** asks whether exhaustive search can be avoided in general.

Finally, we emphasize that the modern definition of **NP** (Definition 7.3) is phrased in a way that captures a widespread philosophical phenomenon: appreciating the correctness of a statement is often easier than coming up with the statement in the first place. There are many examples. For instance, grading an exam seems easier than coming up with the solutions in the first place. Likewise, appreciating a great piece of music is easier than composing it yourself. The **P** vs. **NP** conjecture asks whether this is actually the case.

Problems

Problem 7.11 (Proof of Proposition 7.4). Prove both $\mathbf{P} \subseteq \mathbf{NP}$ (i.e. every problem in **P** is also in **NP**) and $\mathbf{NP} \subseteq \mathbf{EXP}$ (i.e. every problem in **P** is also in **EXP**).

Problem 7.12 (Factoring: efficient decision implies efficient search). Consider the language FACTORING introduced in Eq. (11.1) Suppose you had access to a black box – e.g. a quantum computer that runs Shor's algorithm – that could decide FACTORING in running time $O\left(n^3\right)$, where $n = \lceil \log_2(N) \rceil$ is the bit length of N. Show that this would then also allow you to factorize worst-case instances

$N = M_1 \times M_2$ with $M_1, M_2 \in \mathbb{N}$ prime in time $O(n^3 \log(n))$. **Hint:** binary search.

8. The Cook-Levin Theorem

Date: 24 November 2023

Agenda

Agenda:

1. k-SAT
2. Cook-Levin theorem
3. Proof sketch
4. Implications

Informally speaking, **NP** contains all problems that may be difficult to solve, but are at least easy to check. Many important problems fall into this category, e.g. Factoring, Traveling Salesperson (TSP) and many others. These problems don't seem to have much in common. But today, we will introduce a single **NP** problem that 'rules them all', see Figure 8.1 for a caricature. This problem is 3-SAT – the Boolean satisfiability problem for conjunctive normal formulas (CNFs) with clauses of size (at most) 3. And a seminal result by Cook and Levin shows that *every* problem instance in **NP** can be represented by an instance of

one **NP**-problem to rule them all

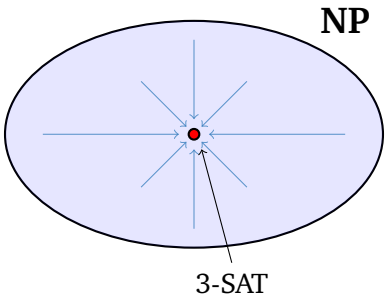


Figure 8.1 *One **NP** problem to rule them all:* 3-SAT, a special case of the Boolean satisfiability problem, is at the center of the problem class **NP**. It is contained in **NP** and every other problem in **NP** can be reduced to an instance of 3-SAT (at a polynomial overhead). This is the content of the Cook-Levin theorem.

3-SAT. So, 3-SAT is, in a precise sense, at least as hard as every other problem in **NP**. The content of this lecture is largely taken from [AB09, Sec. 2.3]

8.1 k-SAT, aka Boolean satisfiability

Recall that a n-variable *Boolean formula* $\varphi(z) = \varphi(z_1, \dots, z_n)$ is comprised of logical operators AND ($\wedge$), NOT ($\neg$) and OR ($\vee$). We associate true with 1 and false with 0 and say that a formula φ is *satisfiable* if there exists (at least) one bitstring $z_\sharp \in \{0,1\}^n$ such that $\varphi(z_\sharp) = 1$.

satisfiability

A Boolean formula is in *conjunctive normal form (CNF)* if it is an AND of OR's of potentially negated variables. For instance,

conjunctive normal form (CNF)

$$\varphi(z_1, z_2, z_3) = \underbrace{(\bar{z}_1 \vee z_2 \vee z_3)}_{\text{clause 1}} \wedge \underbrace{(z_1 \vee \bar{z}_2 \vee z_3)}_{\text{clause 2}} \wedge \underbrace{(z_1 \vee z_2 \vee \bar{z}_3)}_{\text{clause 3}} \tag{8.1}$$

is in CNF. Here, $\bar{z}_i = \neg z_i$ denotes the negation of variable z_i. The OR-expressions in brackets are called clauses (see gray brackets). More generally, a CNF formula looks like

$$\varphi(z) = \varphi(z_1, \dots, z_n) = \bigwedge_{i=0}^{m-1} \underbrace{\left(\bigvee_{j=0}^{k-1} \tilde{z}_{i_j} \right)}_{\text{clauses of size} \le k}, \tag{8.2}$$

where $i_j \in \{0, \dots, n-1\}$ is an index and each $\tilde{z}_{i_j}$ is a placeholder for either z_{i_j} (variable) or $\bar{z}_{i_j}$ (negated variable). The parameter $m \in \mathbb{N}$ counts the total number of clauses, while the other parameter $k \in \mathbb{N}$ counts the maximum size of an individual clause. We say that a CNF formula is a *k-CNF* if the maximum clause size is k. For instance, Eq. (8.1) presents a 3-CNF ($k = 3$) that contains $m = 3$ clauses in $n = 3$ variables.

k-CNF = CNF with maximum clause size k

Example 8.1 (logical equality between bitstrings). The Boolean formula

$$\varphi_{1\text{bit}}(a, b) = \left(a \vee \bar{b}\right) \wedge (\bar{a} \vee b)$$

is a 2-CNF with $n = 2$ variables and $m = 2$ clauses. It evaluates to one if and only if $a = b$ (logical equality). We can readily extend this construction to $2n$ bits. For $x, y \in \{0,1\}^n$, we obtain

$$\varphi_{n\text{bit}}(x, y) = \varphi_{1\text{bit}}(x_0, y_0) \wedge \varphi_{1\text{bit}}(x_1, y_1) \dots \wedge \varphi_{1\text{bit}}(x_{n-1}, y_{n-1}). \tag{8.3}$$

This formula evaluates to one if and only if the two bitstrings are exactly the same. Eq. (8.3) describes a 2-CNF (every clause has size 2) with $2n$ variables and $2m$ clauses. ∎

The *Boolean satisfiability problem* asks whether a given k-CNF is satisfiable. Since we can encode general k-CNF formulas (8.2) φ into bitstrings $\llcorner \varphi \lrcorner \in \{0,1\}^*$, we may reinterpret this decision problem as a language over bitstrings.

Definition 8.2 (k-SAT). The language k-SAT contains all bit encodings of k-CNFs that are satisfiable:

k-SAT = satisfiability of k-CNFs

$$k\text{-SAT} = \{\llcorner \varphi \lrcorner : \ \varphi \text{ is a } k\text{-CNF that is satisfiable}\} \subset \{0,1\}^* . \tag{8.4}$$

It is widely believed that k-SAT $\notin$ **P**, i.e. there is no polynomial-running time algorithm that can determine whether a k-CNF φ is satisfiable for $k \geq 3$ (for $k = 1$ and $k = 2$ we know algorithms that solve k-SAT in polynomial times). In fact, we will ourselves provide strong evidence in Corollary 8.7 below. Instead, k-SAT is a prime example of a problem in **NP**. Finding a solution might be difficult, but checking the correctness of a solution is easy. More precisely, recall the following definition from Lecture 7.

Definition 8.3 (NP, Restatement of Definition 7.3 in Lecture 7**).** A language $A \in \{0,1\}^*$ is in **NP** if there exist polynomials $p, q : \mathbb{N} \to \mathbb{N}$ and a Turing machine M (*the verifier*) with the following two properties:

- **Completeness:** if $x \in A$, then there exists a *short certificate* $y \in \{0,1\}^*$ with $|y| = p(|x|)$ such that $M(x,y) =$ accept after (at most) $q(|x|)$ steps.
- **Soundness:** else if $x \notin A$, then $M(x,y) =$ reject for all possible certificates $y \in \{0,1\}^*$.

Proposition 8.4 k-SAT $\in$ **NP**.[1]

k-SAT $\in$ **NP**

Proof. Suppose that $\llcorner \varphi \lrcorner \in k$-SAT. Then, there must exist at least one bitstring $y = y_0 \cdots y_{n-1}$ such that $\varphi(y) = 1$. Conversely, if $\llcorner \varphi \lrcorner \notin k$-SAT, then $\varphi(y) = 0$ for all inputs $y \in \{0,1\}^n$. This tells us that we can interpret satisfiable assignments $y \in \{0,1\}^n$ as certificates. For verification, we simply evaluate the Boolean formula on such an assignment and check whether

$$M(\llcorner \varphi \lrcorner, y) = \varphi(y) = \begin{cases} 1 & \text{(accept)}, \\ 0 & \text{(reject)}. \end{cases}$$

Note that this TM M behaves the same for all possible inputs and certificates. It always reads in a bit encoding $\llcorner \varphi \lrcorner$ of a Boolean k-CNF and evaluates it on a bitstring y. It is also easy to check that this verification procedure is both complete and sound.

To fully complete the argument, one must also ensure that certificate length $|y|$ and running time of the verification procedure both scale polynomially in the input size $|\llcorner \varphi \lrcorner|$. We leave this as an exercise. ∎

Finally, it is worthwhile to point out that the maximum clause size $k \geq 4$ in a CNF can be reduced to 3 at the cost of extra variables and extra clauses.

[1] Note that this does not contradict the above claim that 1-SAT and 2-SAT are actually contained in **P**. After all, **P** $\subseteq$ **NP**.

Lemma 8.5 Every k-CNF φ can be transformed into a 3-CNF ψ such that φ is satisfiable if and only if ψ is satisfiable. *k-SAT reduces to 3-SAT*

Proof sketch. Let us first show how a 4-clause can be mapped to two 3-clauses that are satisfiable if and only if the original 4-clause is. Suppose, for concreteness, $C = z_0 \vee \bar{z}_1 \vee \bar{z}_2 \vee z_3$. We add a new variable w and divide C (along the middle) into two 3-clauses that also feature w: $C_0 = z_0 \vee \bar{z}_1 \vee w$ and $C_1 = \bar{w} \vee \bar{z}_2 \vee z_3$. It is now easy to check that $C \in$ 4-SAT if and only if $C_1 \wedge C_2 \in$ 3-SAT. The same idea applies to any clause of size $k = 4$.

We can also extend it to decompose a general k-clause into two $(k-1)$-clauses that feature one additional variable. Iterating this procedure allows us to replace k-clauses by an AND of $2(k-3)$ 3-clauses that feature $(k-3)$ extra variables. ■

Before moving on, we emphasize that $k = 3$ is the smallest clause size that is achievable in this way.

8.2 The Cook-Levin Theorem

The difficulty of k-SAT seems to change drastically with maximum clause size k. It is known that 1-SAT and 2-SAT can always be solved in polynomial running time, i.e. 1-SAT, 2-SAT $\in$ **P**. In stark contrast, 3-SAT is the paradigmatic **NP**-problem. If we can solve 3-SAT, we can also solve *every other problem in **NP*** with only polynomial overhead. This is the content of the following deep result that was established independently by Cook (1971) and Levin (1973).

Theorem 8.6 (Cook-Levin theorem). Let $A \subseteq \{0,1\}^*$ be any language in **NP**. Then, we can reformulate the question $x \in A$ into an instance of 3-SAT that only depends on the input $x \in \{0,1\}^*$ and the verifying TM M. More precisely: *Cook-Levin theorem*

(i) if $x \in A$, then the 3-SAT formula $\varphi_{x,M}$ is satisfiable;
(ii) else if $x \notin A$, then the 3-SAT formula $\varphi_{x,M}$ is not satisfiable.

Crucially, the size of $\varphi_{x,M}$ – both in terms of variables and clauses – only depends *polynomially* on the input size $n = |x|$.

It is this central position of 3-SAT within **NP** that has been illustrated in Figure 8.1 above. The following result is an immediate consequence of this distinguished role.

Corollary 8.7 Suppose that 3-SAT $\in$ **P** (i.e. there exists a polynomial-running time algorithm that always decides whether a 3-SAT formula is satisfiable). Then, **P** = **NP**.

8.3 Proof sketch for Theorem 8.6

Rather than presenting a traditional polished proof, we will try to focus on the underlying challenges and solution strategies. In fact, our first two proof

attempts will fail. But understanding how they fail will bring us closer to a working proof argument.

Recall that we can associate a language $A \subseteq \{0,1\}^*$ with the abstract Boolean function

$$f_A(x) = \begin{cases} 1 & \text{if } x \in A, \\ 0 & \text{else if } x \notin A. \end{cases} \tag{8.5}$$

And it seems plausible that we can further decompose such Boolean functions into sequences of elementary logical operations $(\wedge, \vee, \neg)$. In fact, we can even ensure that the resulting logical formula is in CNF.

universality of $\wedge$, $\vee$ and $\neg$

Theorem 8.8 (universality of elementary logical operations). Let $f : \{0,1\}^l \to \{0,1\}$ be a Boolean function on l bits. Then, we can construct a l-CNF φ with (at most) 2^l clauses such that $\varphi(z) = f(z)$ for all $z \in \{0,1\}^l$.

Proof sketch. For every $v \in \{0,1\}^l$, we can easily construct a l-CNF C_v that behaves like $\neg v$, i.e. $C_v(v) = 0$ and $C_v(z) = 1$ for all $z \neq v$. E.g., for $l = 4$ and $v = 1101$, we would set $C_v(z) = \bar{z}_1 \vee \bar{z}_2 \vee z_3 \vee \bar{z}_4$. We can use such clauses and the Boolean function itself to come up with a l-CNF reformulation of $f(z)$:

$$\varphi(z) = \bigwedge_{\substack{v \in \{0,1\}^l : \\ f(v) = 0}} C_v(z) = \begin{cases} 1 & \text{if } f(z) = 1, \\ 0 & \text{else if } f(z) = 0. \end{cases}$$

The problem is that the large AND goes over all bitstrings that evaluate to zero. And, there may be 2^l many of them ('exponential overhead'). ■

However, directly applying Theorem 8.8 only shifts the problem to a different place. If we don't know how to compute f, we don't know how to construct φ. In order to make actual progress, we need to exploit characteristic properties of the problem class **NP**.

For instance, Definition 8.3 asserts that the verification procedure M is fixed and has polynomially bounded running time. Perhaps, the following Boolean function is an easier target:

$$f_M(x,y) = \begin{cases} 1 & \text{if } M(x,y) = \text{accept}, \\ 0 & \text{else if } M(x,y) = \text{reject}. \end{cases} \tag{8.6}$$

Here, there is at least hope that we can convert $f_M(x,y)$ into a k-CNF $\tilde{\varphi}_M(x,y)$. And such a conversion is also promising for a different reason. Note that **NP** languages A are difficult, because we are only given the input x and not the associated certificate y. Input $x \in A$ if and only if a certificate y exists that leads to M accepting the joint input (x,y). But this actually plays nicely with Boolean satisfiability. For fixed input x and fixed verification procedure M, we simply define

$$\tilde{\varphi}_{M,x}(y) = \tilde{\varphi}_M(x,y),$$

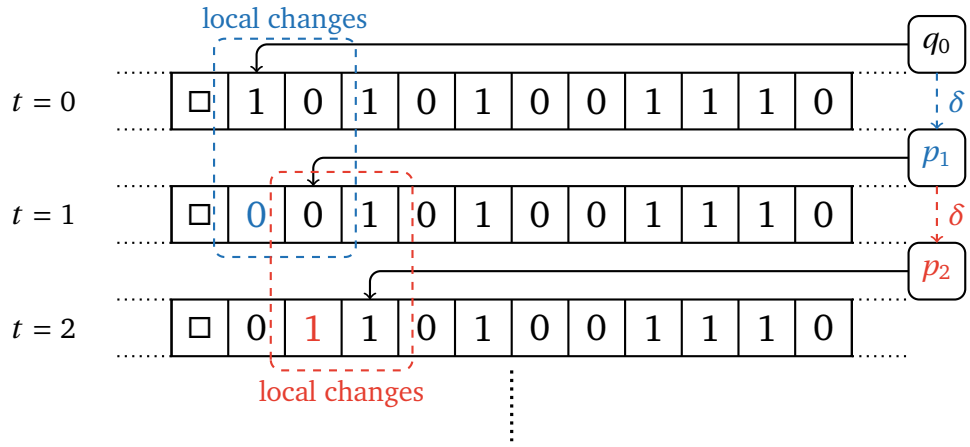


Figure 8.2 *Visualization of TM computation steps:* A TM operates in steps, where each step only involves a very restricted set of operations (read a symbol, change internal state, write a symbol and move left/right). Every such step can only affect the internal state register, as well as a very small portion of the tape. I.e. *TM computations are highly local*.

where $\tilde{\varphi}_M(x,y)$ is a $(|x|+|y|)$-CNF that represents $f_M(x,y)$ from Eq. (8.6). Subsequently, we can even decompose clauses to obtain a 3-CNF $\varphi_{M,x}(y)$ with the same functionality, see Lemma 8.5.

This is where things get interesting. The 3-CNF $\varphi_{M,x}$ is satisfiable if and only if there exists an input y (the certificate) such that $M(x,y) = \text{accept}$. More formally,

$$\varphi_{M,x}(y) \in \text{3-SAT} \qquad \Leftrightarrow \qquad x \in A \quad \text{(whenever } A \in \mathbf{NP}\text{)}.$$

This is tantalizingly close to the statement in Theorem 8.6. But, we are not quite there yet. In fact, a huge problem remains. Even when x is fixed, the Boolean formula (8.6) is still a function on length-m bitstrings, where $m = |y|$. And using Theorem 8.8 to convert it into a m-CNF $\tilde{\varphi}_{M,x}$ may produce as many as 2^m different clauses of length m. This number scales *exponentially* in the number of certificate bits m. And this is a deal breaker! Already writing down all clauses required would take more time (and space) than simply computing $M(x,y)$ for all 2^m posssible certificates $y \in \{0,1\}^m$.

This conundrum can be resolved with an ingenious change of perspective. Instead of representing the logical functionality of a verification procedure as a single huge CNF, we should try to represent the actual TM computation by a sequence of small Boolean formulas – one for each computational step. This allows us to exploit the particular structure of TM computations for **NP** verification:

(i) **Polynomial running time:** Evaluating $M(x,y)$ requires at most poly(n)-many steps, where $n = |x|$. This ensures that the total number of small

Boolean functions – one for each computational step – can only grow polynomially with input size.

(ii) **TM computations are local:** individual TM steps are highly localized. They only affect a single tape square (whose value might change), the underlying state register and a simple movement by the tape head. Illustrated in Figure 8.2, this can be used to ensure that the individual Boolean functions are small in the sense that they only address a small number of variables (bits/truth values) at a time.

These two features allow for transforming entire TM computations – on different inputs – into a single CNF formula. In order to achieve this, we must ensure three qualitatively different things:

1 **Initialization:** The initial tape content must be of the form $xy \in \{0,1\}^{n+m}$, where x denotes the actual input. The other part – which corresponds to the certificate y – remains unconstrained.
2 **Termination:** After $T = \text{poly}(n)$ steps, the TM must transition into the accept state q_{accept}.
3 **Consistent computation:** the computational trajectory between initialization and termination must reflect a valid TM computation. That is, the local changes that occur between step t and $t+1$ must be consistent with the transition function $\delta : Q \times \Gamma \to Q \times \Gamma \times (L, R)$ that specifies M.

Remarkably, we can convert all these conditions into a polynomial number of 3-CNFs. Our final 3-CNF will be an AND of three structurally different CNFs:

$$\varphi_{M,x}(z) = \varphi_{\text{input}}(z) \wedge \varphi_{\text{consistency}}(z) \wedge \varphi_{\text{termination}}(z), \tag{8.7}$$

where z denotes a bitstring of length $\text{poly}(n+m)$. Different aspects of the computation are encoded into different portions of this bitstring, see Figure 8.3 for an illustration.

The first function $\varphi_{\text{input}}(z)$ only affects the first $|n| + |m|$ bits of z (original input) and requires that the first n bits are equal to the **NP**-problem input x. The second batch of m bits remains unconstrained (for now). Enforcing a logical equality between n bits requires $2n$ clauses of size 2, see Example 8.1.

The second function $\varphi_{\text{consistency}}(z)$ is really an AND of T smaller Boolean functions $\varphi_{\text{cons},t}(z)$, one for each computational step. The individual $\varphi_{\text{cons},t}(z)$'s check that each computational step is carried out correctly. The central portions of bitstring z are used to encode internal states of the TMs tape head: $r_0 = q_0, r_1, \ldots, r_{T-1} = q_{\text{accept}}$, as well as tape content changes at current head positions. A constant number of c bits is required for each $t = 0, \ldots, T-1$ and the corresponding Boolean function $\varphi_{\text{cons},t}$ also only involves these c bits. This is important, because it allows us to use Theorem 8.8 to represent each of them as a c-CNF with (at most) $2^c = O(1)$ clauses. The overhead of transforming this into a 3-CNF is also constant. Putting everything together reveals that

$$\varphi_{\text{consistency}}(z) = \varphi_{\text{cons},0}(z) \wedge \varphi_{\text{cons},1}(z) \wedge \cdots \wedge \varphi_{\text{cons},T-1}(z)$$

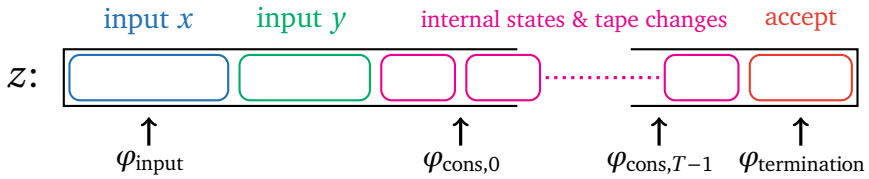


Figure 8.3 *Visualization of a Cook-Levin bitstring encoding:* We encode an entire TM verification procedure into Boolean formulas on a (long) bitstring z. The first portion (blue) is reserved for the actual input x. The second part (green) is unconstrained and provides space for the certificate y, while the final part (red) encodes output and final state. Inbetween are T auxiliary bitstrings (magenta) that allow us to keep track of the T actual TM computation steps. Different Boolean functions enforce different consistency conditions, e.g. using the correct input, performing computational steps correctly and terminating in an accept state. Remarkably, both the length of the bitstring and the number of clauses remains polynomial in $n = |x|$, $m = |y|$ and $T = \text{poly}(n)$.

is a 3-CNF with (at most) $c2^c T = \text{poly}(n)$ clauses (recall that the running time T of M scales polynomially in the original input size $n = |x|$).

The third function is easy by comparison. We simply need to check that the internal state – encoded into the final portion of the bitstring z – has moved to the accepting state. This can be achieved by a single Boolean condition that again involves a constant number of bits.

Putting everything together provides us with a 3-CNF

$$\varphi_{M,x}(z) = \varphi_{\text{input}}(z) \wedge \varphi_{\text{cons},0}(z) \wedge \cdots \wedge \varphi_{\text{cons},T-1}(z) \wedge \varphi_{\text{output}}(z)$$

that affects $|z| = n + m + Tc + c' = \text{poly}(n)$ bits and contains $2n + Tc2^c + 2^c = \text{poly}(n)$ clauses. But, most importantly,

$$\varphi_{M,x}(z) \in \text{3-SAT} \quad \Leftrightarrow \quad \text{there exists } y \in \{0,1\}^m \text{ s.t. } M(x,y) = \text{accept}.$$

This is precisely the content of Theorem 8.6

Warning 8.9 Our arguments are really only a sketch of the main proof idea. We have left many important questions unanswered. E.g. how do we know how large certificate size $m = |y|$ and running time $T(n)$ actually are? Also, how do we actually follow the movement of the tape head throughout the computation? And, finally and perhaps most importantly, how do we actually convert checks for computational consistency into actual Boolean formulas that only involve a few variables at a time? All these things can be sorted out, but it requires diligence and effort. We refer to standard textbooks, e.g. [ABo9], for a more thorough treatment. ∎

8.4 Context and implications

The Cook-Levin theorem is remarkable, because it relates problem types that don't seem to have very much in common. Membership in **NP** is all it takes. In particular, such diverse problems like Factoring, Traveling Salesperson (TSP), the ground state problem in physics, or graph isomorphism can all be mapped to particular instances of 3-SAT – a fundamental problem class in propositional logic that dates back to ancient Greece. What is more, this transformation only incurs a polynomial overhead (in running time and input size) and can often be made explicit. This underscores that it can be very valuable to have friends or colleagues who are very good at solving 3-SAT. They may well be able to help you with any type of **NP**-problem!

3-SAT is at the center of **NP**

The Cook-Levin theorem is also a good example for the power of abstraction. The statement itself holds regardless of the underlying computational model. This is because the TM model is, up to polynomial overhead, equivalent to all (classical) models of computations. But the proof would have been much more difficult to discover in the context of more modern and convenient models, like the python programming language. It is really the mechanical and highly constrained inner working of TMs that has made all the difference here.

power of abstraction

9. Karp reductions and NP-completeness

Date: self study **Lecturer:** Sibylle Möhle (02 December 2021)

Agenda

In the last lecture, we learned that every problem in **NP** can be expressed as (or reduced to) an instance of 3-SAT. However, 3-SAT is not the only problem having this property, but there are many others. We look at a generalization of the notion of this reducibility, namely Karp reducibility.

The content of this lecture is to a certain extent taken from [AB09, Sec. 2.2] and [Kar72], [KS17, Sec. 6], and [Way01, Slides 28–31].

Agenda:

1. Karp reductions
2. **NP**-hardness
3. **NP**-completeness
4. Prove **NP**-completeness
5. Implications

9.1 Karp reductions

In the last lecture, we have seen how k-SAT can be reduced to 3-SAT. We would expect that such reductions exist between other languages as well, and we also do not want to restrict ourselves to the problem class **NP**. In other words, we want a more general notion of reducibility. Richard M. Karp introduced the concept of polynomial-time reducibility back in 1972. In his paper [Kar72], he also provides a list of 21 problems in **NP**, which he calls "complete", since all other problems in **NP** can be reduced to these with mostly polynomial overhead.

Karp reduction

Definition 9.1 (Karp-reducibility). A language $L \subseteq \{0,1\}^\star$ is *polynomial-time Karp reducible* to a language $L' \subseteq \{0,1\}^\star$, denoted $L \leq_p L'$, if there exists a polynomial-time function $f\colon \{0,1\}^\star \to \{0,1\}^\star$ such that for every $x \in \{0,1\}^\star$, we have that $x \in L$ if and only if $f(x) \in L'$.

Basically, a Karp reduction from L to L' maps $x \in L$ to $f(x) \in L'$ and $x \in \overline{L} = \{0,1\}^\star \setminus L$ to $f(x) \in \overline{L'}$ in polynomial time. It can be used to

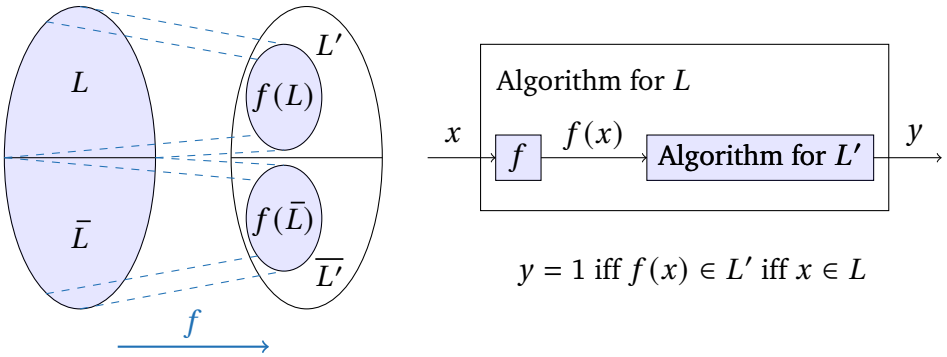


Figure 9.1 Reduction from L to L'. The polynomial-time function f maps elements from L to elements of L' and elements from $\overline{L} = \{0,1\}^\star \setminus L$ to elements from $\overline{L'} = \{0,1\}^\star \setminus L'$. In order to reduce L to L', we construct an algorithm taking as input $f(x)$, which returns 1 iff $f(x) \in L'$ which is the case iff $x \in L$.

transform a polynomial-time TM M' deciding L' into a polynomial-time TM M deciding L by setting $M(x) = M'(f(x))$, $M(x) = 1$ (accept), iff $f(x) \in L'$.

It is important to notice that we start from an algorithm (the TM M') for L' and construct it such that it behaves like an algorithm (the TM M) for L, i.e., it returns 1 iff $x \in L$. This is exactly the case if $f(x) \in L'$ as is visualized in Fig. 9.1.

Why not the other way round? Intuitively, we can model an easy problem as a hard one. But that does not mean that the easy problem is hard per se.

Consider as an example 2-SAT. To make the argument more concrete, let our formula be $F = (a \vee b) \wedge (c \vee \overline{b})$ defined over the set of variables $V = \{a, b, c\}.\}$. In Sec. 8.2 we learned that 2-SAT $\in$ **P**. We can easily build a 3-CNF F' which is satisfiable if and only if F is satisfiable. To this end we introduce new variables x and y and define $F' = (a \vee b \vee x) \wedge (a \vee b \vee \overline{x}) \wedge (c \vee \overline{b} \vee y) \wedge (c \vee \overline{b} \vee \overline{y})$. Now, we have an instance of 3-SAT. Obviously, F and F' are equisatisfiable, i.e., F' is satisfiable if and only if F is satisfiable. However, $F' \in$ **NP**.

The main objective behind reductions is to express a given language by means of another language which is *at least as hard* as the original language. Clearly, this objective is missed by transforming an instance of 2-SAT into an instance of 3-SAT. We will make clear what we mean by *at least as hard* in a moment, when presenting properties of Karp (or polynomial-time) reductions. But let us walk through an example first.

Definition 9.2 (CLIQUE). Given an undirected graph G and a positive integer C, does G contain a subset of size C consisting of mutually adjacent nodes?

Example 9.3 (clique). Figure 9.2 shows the concept of clique, which is defined as a subset of the vertices contained in a graph in which all vertices are pairwise adjacent. ∎

Proposition 9.4 (SAT $\leq_p$ CLIQUE). SAT $\leq_p$ CLIQUE

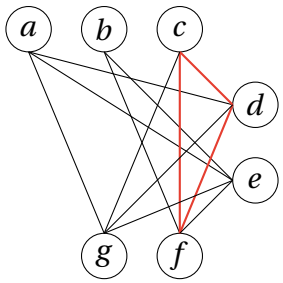


Figure 9.2 Clique. We are given people a, b, c, d, e, f, g and their relationships (a,d), (a,e), (a,g), (b,e), (b,f), (c,d), (c,f), (c,g), (d,f), (d,g), (e,f), (e,g). These relationships are visualized above as a graph G. Two persons knowing each other are connected by an edge. A clique is a subset of the vertices in an undirected graph consisting of pairwise adjacent vertices. As an example, the graph depicted above contains a clique of size $C = 3$ given by $\{c,d,f\}$. These vertices are connected by the edges highlighted in red.

Proof. In our proof we follow [Way01] but with a smaller example. The basic idea is that, given an instance F of SAT, i.e., a Boolean formula F, we want to find a function converting F into a graph G such that each clique of G of size C can be mapped onto a model of F.

Without loss of generality, we define $C = 3$, which also gives us the number of clauses contained in F. Let our formula be

$$F = \underbrace{(\overline{x} \vee y \vee z)}_{C_1} \wedge \underbrace{(\overline{y} \vee \overline{z})}_{C_2} \wedge \underbrace{(\overline{y} \vee x)}_{C_3}.$$

We create a person for every literal as visualized in Fig. 9.3 a); the persons contained in one clause are grouped together. Now we are going to create relationships between these persons. We say that all persons know each other except if the literals representing them are either contained in the same clause or are negations of each other. We connect people knowing each other by an edge according to Fig. 9.3 b) obtaining a graph representation of F. This graph contains, for instance, the clique c_1 highlighted in red in Fig. 9.3 c), from which we can construct a model m_1 of F (or the other way round). It suffices to set the literals representing the people in c_1 to 1 (or to add the persons represented by the literals in the model to c_1). In fact, the assignment $m_1 = x\,\overline{y}\,z$ satisfies F.

Notice that according to Def. 9.1 *every* clique in G must be mapped onto a model of F (and vice versa). In our example, we additionally observe that two different cliques of G can be mapped to the same model of F. Consider the clique c_2 consisting of the vertices representing the literals z, $\overline{y}$, and $\overline{y}$ contained in C_1, C_2, and C_3, respectively. It contains no vertex representing one of x and $\overline{x}$, i.e., the value of x is irrelevant for the model to which c_2 is mapped. By setting x to 1, we obtain again m_1. If instead we set x to 0, we obtain the model $m_2 = \overline{x}\,\overline{y}\,z$. This model is also mapped onto by two cliques,

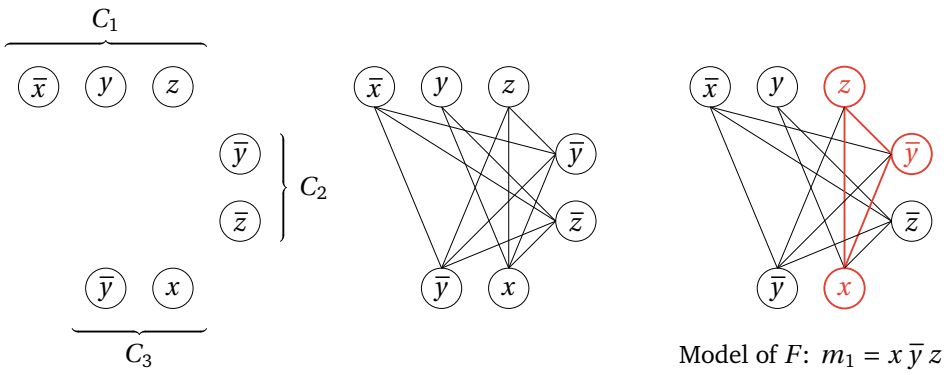


Figure 9.3 Reducing SAT to CLIQUE. Suppose we have $F = C_1 \wedge C_2 \wedge C_3$ and clique size $C = 3$. a) We create a person for each literal in the formula (left hand side). b) Two persons know each other if either they are not represented by literals contained in the same clause or they are not represented by literals which are the negation of each other. Persons knowing each other are connected by an edge. c) The resulting graph contains a clique of size 3 represented by the literals highlighted in red and connected by red edges. Setting those literals to 1 gives us a satisfying assignment for F. Notice that clique size and number of clauses coincide.

namely c_2 and the clique c_3 consisting of the vertices represented by $\overline{x}$, $\overline{y}$, and $\overline{y}$ contained in C_1, C_2, and C_3, respectively. ■

Exercise 9.5 Determine all cliques of G and the models of F they are mapped onto for the formula in the proof of Prop. 9.4.

9.2 Properties of Karp reductions

In this section, we are going to present some useful properties of Karp or polynomial-time reductions. In Sect. 9.3, we will show properties, which are useful in the context of **NP**-completeness.

Lemma 9.6 (Determining P-membership). Let $L, L' \in \{0,1\}^*$ be two languages. If $L \leq_p L'$ and $L' \in \mathbf{P}$, then $L \in \mathbf{P}$. reduction to **P**

Proof. We can check whether $x \in L$ in polynomial time: Given a polynomial-time function f, we first compute $f(x)$, and then check in polynomial time whether $f(x) \in L'$. ■

Lemma 9.6 clarifies, why we say that L' is *at least as hard* as L. It provides a means to determine whether, in this special case, a language is in **P**.

Theorem 9.7 (Transitivity of Karp reductions). If $L \leq_p L'$ and $L' \leq_p L''$, then $L \leq_p L''$. transitivity

Proof. We make use of the following observation: If p and q are two functions growing at most as n^c and n^d, respectively, their composition $p(q(n))$ grows at most n^{cd}, which is also polynomial.

Suppose f_1 is a polynomial-time reduction from L to L' and f_2 is a polynomial-time reduction from L' to L''. Then the mapping $x \to f_2(f_1(x))$ is a polynomial-time reduction from L to L'', since $f_2(f_1(x))$ takes polynomial time. Finally, $f_2(f_1(x)) \in L''$ iff $f_1(x) \in L'$, which holds iff $x \in L$. ■

9.3 NP-completeness

Recall that a language L is in **NP**, if there exists an efficient method to verify membership. In other words, if we are given some x and told that $x \in L$, we can verify this claim with a polynomial overhead. There might or might not exist an efficient method to find some $x \in L$. The first case captures the class of problems in **P** introduced in Sect. 6.4. In this lecture, we are interested in the second type of problems, i.e., the problems which are (probably) exponentially hard to solve but the correctness of whose solutions can be checked efficiently. In the last lecture we saw that 3-SAT 'rules all problems in **NP**'. In this section we are going to precise what that exactly means and show that there are other problems in **NP** having this property as well.

Definition 9.8 (NP-hardness and NP-completeness). We say that L' is **NP**-*hard* if $L \leq_p L'$ for every $L \in$ **NP**. We say that L' is **NP**-*complete* if $L' \in$ **NP** and L' is **NP**-hard. **NP**-hard **NP**-complete

Proposition 9.9 (NP-completeness of 3-SAT). 3-SAT is **NP**-complete. 3-SAT is **NP**-complete

Proof. We show that (a) 3-SAT $\in$ **NP** and (b) 3-SAT is **NP**-hard, i.e., that every $L \in$ **NP** can be mapped by an instance of 3-SAT, or $L \leq_p$ 3-SAT. As for (a), this is exactly Prop. 8.4, which we already proved, while (b) is Theorem 8.6. By Def. 9.8, 3-SAT is **NP**-complete. ■

The following lemma is a consequence of Theorem 8.6 and Prop. 9.9.

Lemma 9.10 If $L \in$ **NP**, then $L \leq_p$ 3-SAT.

Proposition 9.11 CLIQUE is **NP**-complete. CLIQUE is **NP**-complete

Proof. We first need to show that (a) CLIQUE $\in$ **NP** and then (b) choose an **NP**-hard (or **NP**-complete) problem P and provide a polynomial-time reduction from it to CLIQUE. (a) We can check whether a certificate consisting of C nodes of G is a clique in polynomial time. This establishes that CLIQUE $\in$ **NP**. (b) We choose to reduce from 3-SAT to CLIQUE, i.e., to show 3-SAT $\leq_p$ CLIQUE. The proof is similar to the one in the proof of Prop. 9.4, but restricted to 3-SAT. Since $L \leq_p$ 3-SAT for all $L \in$ **NP** and 3-SAT $\leq_p$ CLIQUE, due to Theorem 9.7

also $L \leq_p$ CLIQUE for all $L \in \mathbf{NP}$, i.e., CLIQUE is **NP**-hard. Together with (a) and Def. 9.8, we obtain that CLIQUE is **NP**-complete. ■

So, 3-SAT is not the only **NP**-complete problem. This means that in order to show that L is **NP**-complete, we could as well reduce CLIQUE to L. And Cook and Levin established **NP**-completeness for many more hard problems.

9.4 A method for proving NP-completeness

We have proved that 3-SAT and CLIQUE are **NP**-complete. These proofs give us a general method to show **NP**-completeness for any problem in **NP**:

Proving **NP**-completeness

Suppose we want to prove that a language Q is **NP**-complete.

1. Show that $Q \in \mathbf{NP}$.
2. Choose some problem P known to be **NP**-hard. Provide a function f transforming any instance of P into an instance of Q such that $x \in P$ iff $f(x) \in Q$.

The direction, in which the reduction is applied, is essential, as explained in detail in Kingsford and Slater [KS17], Section 6.

9.5 Implications

We state some important implications which guide us towards one of the most important questions in computer science: **NP** = **P**?.

Theorem 9.12 If language L is **NP**-hard and $L \in \mathbf{P}$, then **NP** = **P**. If language L is **NP**-complete, then $L \in \mathbf{P}$ if and only if **P** = **NP**.

Corollary 9.13 **P** = **NP** iff 3-SAT $\in$ **P**.

Proof. Suppose 3-SAT $\in$ **P**. Then, due to Prop. 9.10, each $L \in \mathbf{NP}$ is in **P**, hence **NP** = P. If 3-SAT $\notin$ **P**, then clearly 3-SAT $\in$ **NP** and **P** $\neq$ **NP**. ■

If we could find a polytime algorithm for solving 3-SAT, we could conclude that **P** = **NP**. The general belief is that such an algorithm does not exist. However, a proof is still missing.

10. Space complexity

Date: 01 December 2022

Agenda

In the previous lectures, we have focused on analyzing the number of steps required to solve different types of problems (*running time*). Today, we switch focus and talk about space-bounded computations (*memory*). We will see that polynomial space/memory requirements seem to be less stringent than for polynomial running time. The resulting problem class, called **PSPACE**, encompasses both **P** and **NP**, see Figure 11.1. The essence of this large complexity class is captured by *quantified Boolean formulas* and has an appealing interpretation as the quest for finding optimal strategies in 2-player games.

Agenda:

1. Space-bounded computations
2. Relations to other complexity classes
3. **PSPACE**-completeness
4. Optimal strategies for 2-player games

10.1 Space-bounded computations

In this chapter, we consider the complexity of computational problems in terms of the amount of space, or memory, they require. Time and space are two of the most important cost factors when it comes to computational tasks.

Formally, we shall stick to the Turing Machine (TM) model of computation. Similar to running time – which we defined to be the maximum number of computational steps – TMs also provide us with a simple notion of space. It is the maximum number of tape cells occupied throughout the course of a computation.

Definition 10.1 (space-bounded computation). Fix a function $f : \mathbb{N} \to \mathbb{N}$. We say that a language $A \subseteq \{0,1\}^*$ is contained in **SPACE** $(f(n))$ if there is a TM M that decides $x \in A$ vs. $x \notin A$ based on a maximum number of $O(f(|x|))$ non-blank tape symbols.

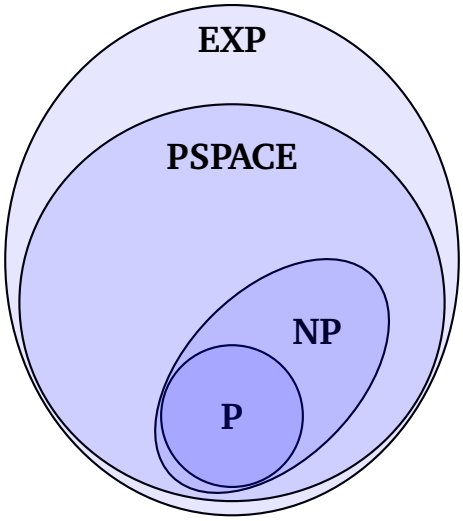


Figure 10.1 *Landscape of complexity classes (to be continued):* The class **PSPACE** is a large complexity class that encompasses all decision problems (languages) that can be decided in polynomial space. **PSPACE** contains both **P** and **NP** and is itself contained in **EXP** – the class of problems that can be decided in exponential time.

This definition is analogous to our definition of time-bounded computations in Lecture 6. Recall that we say that a language $A \subseteq \{0,1\}^*$ is contained in $\mathbf{DTIME}(f(n))$ if there exists a TM M that decides $x \in A$ in worst-case running time $O(f(|x|))$. It is easy to see that space-bounded computations are at least as powerful as time-bounded ones.

Lemma 10.2 **($\mathbf{DTIME}(f(n)) \subseteq \mathbf{SPACE}(O(f(n)))$).** Let $f : \mathbb{N} \to \mathbb{N}$ be a super-linear function, i.e. $n = O(f(n))$, and let $A \subseteq \{0,1\}^*$ be a language. Then $A \in \mathbf{DTIME}(f(n))$ implies $A \in \mathbf{SPACE}(f(n))$.

We show this inclusion only for functions $f(n)$ that scale at least linearly in input size, i.e. $n = O(f(n))$. This is because sub-linear running times don't make sense in our computational model. After all, we do have to read the full input at some point. In stark contrast, sub-linear space is not a deal breaker (e.g. if we consider a TM with a read-only input tape and a much shorter work tape). Instead, sub-linear space restrictions accurately capture situations where the working memory is much smaller than overall data size. Think of web crawlers, streaming algorithms, or external hard drives. Alas, we won't have time to cover this interesting topic and have to refer the interested reader to standard textbooks like [AB09, Sec. 4.4] and [Sip97, Sec. 8.4].

Proof of Lemma 10.2. Fix an input $x \in \{0,1\}^*$ with length $n = |x|$. Then, the assumption $A \in \mathbf{DTIME}(f(n))$ ensures that we can decide $x \in A$ by running a TM for (at most) $O(f(n))$ steps. At each step, the TM can change the content of at most one tape cell. And, since we start with n cells already occupied,

the maximum number of occupied cells can only grow to $n + O(f(n)) = O(f(n))$ (the last equality requires $n = O(f(n))$). This establishes $A \in \mathbf{SPACE}(O(f(n)))$. ∎

Lemma 10.2 tells us that we can solve every polynomial running time problem in polynomial space. The following example indicates that polynomial space also covers some problems for which we don't know any polynomial running time algorithms.

Example 10.3 (3-SAT $\in$ SPACE (n^2)). Let φ be a 3-CNF with n variables and m clauses of size 3 each. The 3-SAT problem asks whether there exists a satisfiable assignment $x = x_0 \cdots x_{n-1}$ such that $\varphi(x) = 1$. Computing $\varphi(x')$ for any given input $x' \in \{0,1\}^n$ is easy and can be achieved in running time (at most) $O(3nm) = O(nm) = O(|\llcorner \varphi \lrcorner|^2)$. What makes the satisfiability problem hard is that there are up to 2^n different inputs that want to be checked.

But, this is a problem for running time, not space. Indeed, we can re-use tape space to sequentially compute $\varphi(x')$ for all possible inputs $x' \in \{0,1\}^n$. All we need is an extra 'counting register' that allows us to keep track of the bitstrings we have already checked. A total of $n = O(|\llcorner \varphi \lrcorner|)$ extra bits suffice for this task. ∎

10.2 The problem class PSPACE

We have seen that space-bounded computations are at least as expressive as time-bounded computations (Lemma 10.2) and can even look much more powerful (Example 10.3). This intuition extends to the union of all decision problems (languages) that can be decided in polynomial space.

Definition 10.4 (PSPACE). The problem class **PSPACE** contains all languages that can be decided using polynomial space (in input size). More formally,

PSPACE definition

$$\mathbf{PSPACE} = \bigcup_{k \geq 1} \mathbf{SPACE}\left(n^k\right), \tag{10.1}$$

where $\mathbf{SPACE}\left(n^k\right)$ has been introduced in Definition 10.1.

PSPACE is the space-centered counterpart of the problem class **P** which encompasses all problems that can be solved in polynomial running time. The following inclusion is an immediate consequence of Lemma 10.2 and Eq. (10.1).

Corollary 10.5 ($\mathbf{P} \subseteq \mathbf{PSPACE}$). Every language $A \subseteq \{0,1\}^*$ that can be decided in polynomial time (in input size) can also be decided in polynomial space (in input size). In formulas: $A \in \mathbf{P}$ implies $A \in \mathbf{PSPACE}$.

In fact, we can even show something stronger. Recall that the problem class **NP** encompasses all languages $A \subseteq \{0,1\}^*$ where membership can be efficiently verified. More formally, there exists a polynomial-running time TM M ('the verifier') such that

(i) if $x \in A$, then there exists $y \in \{0,1\}^*$ with $|y| = \text{poly}(|x|)$ (a 'short certificate') such that $M(x,y) = 1$;

(ii) else if $x \notin A$, then $M(x,y) = 0$ for all $y \in \{0,1\}^*$ with $|y| = \text{poly}(|x|)$ ('no false positives').

Importantly, each run of the verification procedure is efficient. And we can re-use tape space to sequentially loop over all possible certificates $y \in \{0,1\}^{\text{poly}(|x|)}$ and see if one of them prompts the verifier to output $M(x,y) = 1$. This exhaustive search is similar to Example 10.3 and implies the following inclusion.

Proposition 10.6 (NP ⊆ PSPACE). Every language $A \subseteq \{0,1\}^*$, where membership can be efficiently verified, can also be decided in polynomial space. In formulas: $A \in$ **NP** implies $A \in$ **PSPACE**. **NP ⊆ PSPACE**

We start to see a pattern emerging. Polynomial space may seem like a serious restriction. But re-using tape space still allows us to sequentially compute 'simple' functions for exponentially many inputs. And this is enough to handle difficult problems, like 3-SAT (or all of **NP**), where the best-known running time scales exponentially in input size. The following statement establishes a converse relation.

Proposition 10.7 (PSPACE ⊆ EXP). Every language $A \subseteq \{0,1\}^*$ that can be decided in polynomial space, can also be decided in exponential time. In formulas: $A \in$ **PSPACE** implies $A \in$ **EXP**. **PSPACE ⊆ EXP**

Proof. We assume without loss of generality that the TM M that decides A has a binary input alphabet ($\Sigma = \{0,1\}$) and works with a total of $q = |Q| = O(1)$ states. Also, implicit in the definition of **PSPACE** is the requirement that M halts on all inputs x and outputs either $M(x) =$ accept (if $x \in A$) or $M(x) =$ reject (if $x \notin A$).

Now, let x be an input of size $n = |x|$. The assumption $A \in$ **PSPACE** ensures that there is a polynomial $p : \mathbb{N} \to \mathbb{N}$ such that $M(x)$ uses at most $p(n)$ non-blank tape cells throughout the entire computation. Since $\Gamma = \{0,1,\square\}$, this amounts to at most $3^{p(n)}$ distinct tape configurations that could appear in principle. Also, the tape head could be in one of $p(n)$ places while the internal state control may assume one out of $q = |Q| = O(1)$ states. Putting everything together, there can be at most $T(n) = qp(n)3^{p(n)} = 2^{O(p(n))}$ different TM configurations.

We are now in a situation similar to the pumping Lemma from Lecture 4. If we run the TM M for more than $T(n)+1$ steps, then at least one configuration must be visited twice. And this will introduce a loop that cannot be escaped, because M is deterministic. In other words: if M were to run for more than $T(n)+1$ steps, then it must enter a loop and cannot halt. This, however, contradicts one of the defining properties of **PSPACE**. The only resolution is that the running time can never exceed $T(n)+1 = 2^{O(p(n))}$ in the first place. And this, in turn, asserts $A \in$ **DTIME** $\left(2^{O(p(n))}\right) \subseteq$ **EXP**.

■

10.3 The problem class NPSPACE

It is instructive to view **PSPACE** as the space-complexity analogue of **P**, the class of problems that deterministic TMs can solve in polynomial running time. Also, recall that one can define the problem class **NP** via nondeterminism. It is the class of problems that nondeterministic TMs can solve in polynomial running time.

This analogy suggests to introduce a space-complexity analogue to **NP**: the class of problems that a nondeterministic TM can solve in polynomial space (with the extra condition that all computational trajectories halt). More formally, we say that $A \in \mathbf{NSPACE}\left(f(n)\right)$, if there is a nondeterministic TM that decides $x \in A$ based on trajectories that all use at most $O(f(|x|))$non-blank tape symbols. In full analogy to Definition 10.4, we define **NPSPACE**

$$\mathbf{NPSPACE} = \bigcup_{k\geq 1} \mathbf{NSPACE}\left(n^k\right).$$

The space-complexity analogue of the famous $\mathbf{P} \neq \mathbf{NP}$ conjecture is

$$\mathbf{PSPACE} \stackrel{?}{\neq} \mathbf{NPSPACE}.$$

Remarkably, this question has already been resolved in 1970.

Theorem 10.8 (Savitch's theorem). Fix $p : \mathbb{N} \to \mathbb{N}$ such that $p(n) \geq \log(n)$. Then, $L \in \mathbf{NSPACE}\left(p(n)\right)$ implies $L \in \mathbf{SPACE}\left(p(n)^2\right)$.
In particular, $\mathbf{NPSPACE} = \mathbf{PSPACE}$. **NPSPACE = PSPACE**

In words, the overhead of replacing a nondeterministic space-bounded computation by another deterministic computation is at most quadratic in space. Unfortunately, we won't have time to cover the illuminating proof and must defer the interested reader to [AB09, Sec. 4.3].

10.4 PSPACE completeness and a PSPACE complete problem

We strongly believe that **PSPACE** is strictly larger than **P**. In particular, $\mathbf{P} = \mathbf{PSPACE}$ would imply $\mathbf{P} = \mathbf{NP}$, courtesy of Proposition 10.6. But, currently we don't have a formal proof. So the next best thing is to isolate certain problems that capture the essence of **PSPACE**. This can be achieved by means of reductions. Recall the concept of a *polynomial running time (Karp) reduction* from Lecture 9: We write $A \leq_p B$ if there is a polynomial-running time function $f : \{0,1\}^* \to \{0,1\}^*$ such that $x \in A$ if and only if $f(x) \in B$.

Definition 10.9 (PSPACE-hardness and PSPACE-completeness). We say that a language A is **PSPACE**-hard if $A' \leq_p A$ for every $A' \in \mathbf{PSPACE}$. We say that A is **NP**-complete if $A \in \mathbf{PSPACE}$ and A is **PSPACE**-hard.

PSPACE-complete problems are the space-constrained counterpart of **NP**-complete problems, like 3-SAT. (This comparison makes sense, because $\mathbf{NPSPACE} = \mathbf{PSPACE}$ according to Theorem 10.8.)

Arguably, the most interesting **PSPACE**-complete problems again involve *Boolean formulas*. But, now the Boolean formulas are *quantified* using ∃ ('there exists') and ∀ ('for all'). A quantified Boolean formula over truth variables (bits) $x_0, \ldots, x_{n-1} \in \{0,1\}$ has the form

quantified Boolean formula (QBF)

$$Q_0 x_0 Q_1 x_1 \cdots Q_{n-1} x_{n-1} \varphi(x_0, \ldots, x_{n-1}), \tag{10.2}$$

where each Q_i is either ∃ or ∀ and $\varphi : \{0,1\}^n \to \{0,1\}$ is a Boolean formula. We read quantifiers from the left towards the right. That is, $Q_0 x_0$ binds before $Q_1 x_1$, etc. If all variables are quantified, as is the case in Eq. (10.2), then there are no free variables left and the entire expression is either *true* (1) or *false* (0).

Example 10.10 (quantified logical equalities). Recall that we can express *logical equality* between two bits x and y as a 2-CNF

$$\varphi_{1\text{bit}}(x,y) = (x \vee \bar{y}) \wedge (\bar{x} \vee y) = \begin{cases} 1 & \text{if } x = y, \\ 0 & \text{else if } x \neq y. \end{cases}$$

There are four inequivalent ways to quantify this Boolean formula. One of them is

$$\forall x \exists y \, \varphi_{1\text{bit}}(x,y) = \forall x \exists y \, (x \vee \bar{y}) \wedge (\bar{x} \vee y) \, .$$

This QBF claims that for every $x \in \{0,1\}$ there exists a $y \in \{0,1\}$ such that $x = y$ ($\varphi_{1\text{bit}}(x,y) = 1$). This is, of course, true and we write

$$\forall x \exists y \, (x \vee \bar{y}) \wedge (\bar{x} \vee y) = 1.$$

Another way to quantify logical equality is

$$\exists x \forall y \, \varphi_{1\text{bit}}(x,y) = \exists x \forall y \, (x \vee \bar{y}) \wedge (\bar{x} \vee y) \, . \tag{10.3}$$

This formula claims that there exists a bit $x \in \{0,1\}$ such that $x = y$ ($\varphi_{1\text{bit}}(x,y) = 1$) for all $y \in \{0,1\}$. This is, of course, false and we write

$$\exists x \forall y \, (x \vee \bar{y}) \wedge (\bar{x} \vee y) = 0. \tag{10.4}$$

Comparing Eq. (10.3) with Eq. (10.4) underscores that the ordering of quantifiers can make a huge difference. The remaining two quantified logical bit equalities amount to

$$\begin{aligned} \exists x \exists y \, \varphi_{1\text{bit}}(x,y) &= \exists x \exists y \, (x \vee \bar{y}) \wedge (\bar{x} \vee y) = 1 \quad \text{('true')}, \\ \forall x \forall y \, \varphi_{1\text{bit}}(x,y) &= \forall x \forall y \, (x \vee \bar{y}) \wedge (\bar{x} \vee y) = 0 \quad \text{('false')}. \end{aligned}$$

This reflects the basic intuition that ∀-quantifiers are harder to satisfy than ∃-quantifiers. ∎

QBFs also capture and generalize the satisfiability problem.

Fact 10.11 (SAT as QBF). Consider a Boolean formula $\varphi : \{0,1\}^n \to \{0,1\}$ (not necessarily in CNF). Then, this formula is satisfiable ($\varphi \in$ SAT) if and only if a related QBF evaluates to true:

$$\exists x_0 \exists x_1 \cdots \exists x_{n-1}\, \varphi(x_0, \ldots, x_{n-1}) = 1.$$

■

This observation highlights that deciding whether a general QBF is true is at least as hard as solving the satisfiability problem. This problem deserves a proper definition.

Definition 10.12 (TQBF). The *True Quantified Boolean Formula problem* (TQBF) asks whether a fully quantified Boolean formula $Q_0 x_0 \cdots Q_{n-1} x_{n-1}\, \varphi(x_0, \ldots, x_{n-1})$ evaluates to 1 ('true'). The associated language TQBF $\in \{0,1\}^*$ contains all bit encodings of quantified Boolean formulas that evaluate to true.

True Quantified Boolean Formula problem (TQBF)

Note that Fact 10.11 immediately implies that TQBF is **NP**-hard (because we can reduce 3-SAT to it), but it may not be contained in **NP**. It is, however, contained in **PSPACE**.

Proposition 10.13 The language TQBF from Definition 10.12 is contained in **PSPACE**.

Proof sketch. We need to provide a polynomial-space algorithm that decides a QBF (10.2) with n variables and m clauses of constant size each. First, note that if each of the n variables were fully specified, then we could compute the value of $\varphi(x_0, \ldots, x_{n-1})$ in space (and time) $O(nm)$. The actual algorithm works recursively and iteratively specifies assignments to arrive at this case. For starters, note that there are two choices for the first bit: $x_0 = 0$ and $x_0 = 1$. Each such choice produces a formula $\varphi_{x_0=b}(x_1, \ldots, x_{n-1})$ with $b = 0, 1$ that only involves $(n-1)$ variables. And if we had an algorithm A that could decide TQBF for $(n-1)$ variables, then running it once on $\varphi_{x_0=0}(x_1, \ldots, x_{n-1})$ and once on $\varphi_{x_1=1}(x_1, \ldots, x_{n-1})$ would allow us to handle both an $\exists$-quantifier (output 1 if either $A(\varphi_{x_0=0}) = 1$ or $A(\varphi_{x_0=1}) = 1$) and a $\forall$-quantifier (output 1 if both $A(\varphi_{x_0}) = 1$ and $A(\varphi_{x_1}) = 1$). Importantly, we can carry out both algorithm invocations sequentially on the same tape space. After computing $A(\varphi_{x_0})$, we only need to store one bit and can use the remaining space to compute $A(\varphi_{x_1})$.

And this is a promising start for a recursive algorithm. We can keep on specifying variables one-by-one to obtain QBFs with fewer and fewer quantified variables, up to the point where all variables are fully assigned. Let $s_{n,m}$ denote the space algorithm A uses on formulas with n variables and m clauses. Then, a recursive construction that re-uses space ensures $s_{n,m} = s_{n-1,m} + O(nm)$, where $O(nm)$ bounds the cost of specifying $\varphi_{x_1=b}$ for $b = 0, 1$. Repeatedly executing this recursion for all n variables produces a total space complexity of $O(n^2 m)$ which is (at most) quadratic in input size. ■

Theorem 10.14 Every language $L \in$ **PSPACE** can be polynomial-time reduced to an instance of the TQBF problem.

Because **NPSPACE** = **PSPACE**, we may view this result as the space-complexity analogue of the Cook-Levin theorem from Lecture 8. Recall that Cook and Levin showed that every problem in **NP** can be polynomial-time reduced to an instance of the satisfiability problem (or even 3-SAT). Similar in spirit, Theorem 10.14 instead identifies the TQBF problem – a strict generalization of SAT – as the 'one **PSPACE** problem to rule them all'. Unfortunately, we won't have time to provide a proof sketch for Theorem 10.14. We leave it as a challenging, but illuminating, problem for self-study (e.g. by following the arguments provided in [AB09, Proof of Theorem 4.11]). Instead, we conclude this section by combining the implications of Proposition 10.13 and Theorem 10.14.

Corollary 10.15 The TQBF problem is **PSPACE**-complete.

TQBF is **PSPACE**-complete

10.5 The essence of PSPACE: optimal strategies for 2-player games

At first sight, the TQBF problem $Q_0 x_0 \cdots Q_{n-1} x_{n-1}\, \varphi(x_0, \ldots, x_{n-1}) \stackrel{?}{=} 1$ may seem rather abstract and devoid of a captivating intuitive explanation. But, we can change that by slightly massaging the expressions involved. By including new dummy variables in the original formula (such that every choice evaluates to true), we can replace a general QBF by another one where $\exists$ and $\forall$-quantifiers always act one after the other. If the total number of (new) variables $\tilde{n}$ is even, we obtain a QBF of the form

$$\exists y_0 \forall y_1 \exists y_2 \forall y_3 \cdots \exists y_{\tilde{n}-2} \forall y_{\tilde{n}-1}\, \tilde{\varphi}(y_0, \ldots, y_{\tilde{n}-1}). \tag{10.5}$$

(The odd case is similar, but would require slightly different notation). This sorted formula contains at most twice as many variables ($n \leq \tilde{n} \leq 2n$) and is logically equivalent to our original QBF. It also admits a beautiful interpretation as an *optimal strategy in a 2-player game* (with perfect knowledge).

TQBF = optimal winning strategy in 2-player game

The game involves two players, say yours truly (Richard Kueng) and Nina Brandl (the head assistant). Nina controls odd variables, namely $y_0, y_2, \ldots, y_{\tilde{n}-2} \in \{0, 1\}$, while Richard controls even variables, i.e. $y_1, y_3, \ldots, y_{\tilde{n}-1} \in \{0, 1\}$. Both players assign truth variables in an alternating fashion for a total of $\tilde{n}/2$ rounds. Nina wins if the final (fully quantified) Boolean formula evaluates to true. Richard will try everything to thwart that. He wins, if the final formula evaluates to false.

We can now interpret the question whether the QBF (10.5) is true as the question whether Nina can come up with an optimal strategy that lets her win the game regardless of whatever Richard is doing. To see this, it is instructive to look at the quantifiers one at a time. The first quantifier is $\exists$ and addresses the first of Nina's variables. It asks whether there exists a first move Nina can make such that the remaining (partially quantified) formula still has the chance to evaluate to 1 (Nina's win condition). The second quantifier is $\forall$ and

captures all of Richard's possible first moves simultaneously. If the resulting partially quantified Boolean formula can still evaluate to one, then this means that there is nothing that Richard can do (for now) to stop Nina from winning. These alternating steps between Nina and Richard repeat until all variables are assigned. At this point, the game is over and Nina wins if the final formula evaluates to 1 ('true'). And Nina has a winning strategy (with 100% success rate) if and only if

$$\exists y_0 \forall y_1 \exists y_2 \forall y_3 \cdots \exists y_{\tilde{n}-2} \forall y_{\tilde{n}-1} \tilde{\varphi}(y_0, \ldots, y_{\tilde{n}-1}) = 1.$$

There exists ('$\exists$') a sequence of Nina's game move such that the final formula evaluates to true for all ('$\forall$') possible moves of Richard.

In this analogy, the 'board' on which Nina and Richard play is a Boolean formula $\tilde{\varphi}$ whose free variables are alternately assigned by the two competing players. The number of turns is also fixed and depends on the size of the formula. And we just saw that deciding whether one of the players has an optimal winning strategy is at least as hard as solving the TQBF problem. But, we can also always answer this question in polynomial space by repeatedly using tape space to play through all possible game constellations. This observation provides an appealing alternative formulation of Corollary 10.15

Fact 10.16 Determining whether the 2-player QBF game introduced above has an optimal winning strategy is **PSPACE** complete. ∎

The QBF game seems to draw a connection between **PSPACE** on the one hand, and winning strategies of 2-player games on the other. But, keep in mind that complexity-theoretic obstacles – we believe that **PSPACE**-complete problems are *very* difficult – and the entire formalism behind only start to make sense if we increase input size, aka the 'size of the board', to very large numbers. A 8×8 chess board, or a a 19×19 Go grid are far too small to enter this regime. But, it is possible to study generalizations of these games to $n \times n$ boards, where n can become arbitrarily large.

As a concluding remark, we point out that unless **PSPACE** = **NP** (which is widely believed to be false), then there cannot be short certificates for optimal winning strategies. This is in stark contrast to most of the difficult (**NP**-hard) problems we have seen so far.

11. co-NP and the polynomial hierarchy

Date: 14 December 2023

Agenda

Today, we will introduce the problem class **co-NP**. Conceptually, this class is related to **NP**, but subtly different. In fact, it is widely believed that **co-NP** $\neq$ **NP**, although we don't have a formal proof. Viewed as complementary problems, **NP** and **co-NP** form the basis of an entire hierarchy of complexity classes that become ever more complicated. This is the so-called *polynomial hierarchy*. We refer to Fig. 11.1 for a visual illustration, where the polynomial hierarchy is only alluded to (to maintain readability).

A reasonable assumption about this polynomial hierarchy is that it does not collapse. Many interesting complexity-theoretic arguments follow from it. One of them is the observation that integer factorization is (probably) not quite as hard as other problems in **NP**, e.g. 3-SAT or TSP.

Agenda:

1. Motivation: Factoring
2. The problem class **co-NP**
3. The polynomial hierarchy (**PH**)

11.1 Motivation: Factoring is special

To motivate the somewhat advanced topics of today's lecture, we take another look at the decision problem version of (prime) factoring:

$$\textsc{Factoring} = \{\llcorner (N,k) \lrcorner : \ N \in \mathbb{N} \text{ has a prime factor} \leq k\} \subseteq \{0,1\}^* . \tag{11.1}$$

(Here, we do in addition require that the factors be prime numbers.) We do believe that deciding $x \in \textsc{Factoring}$ is difficult in the worst case. We currently don't know any (conventional) algorithms that can factor all integers in polynomial time. Despite its intrinsic difficulty, factoring also has a very desirable property. Once we have found a valid factorization, it is easy to

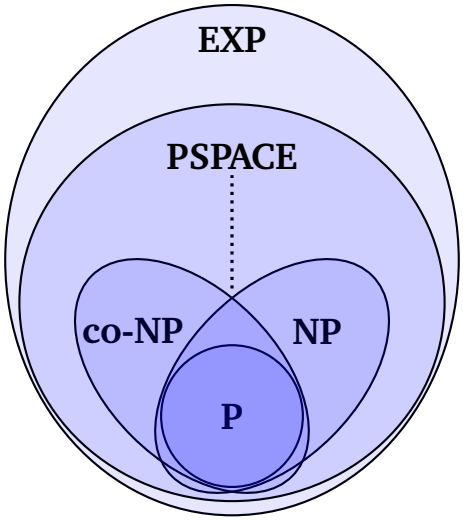


Figure 11.1 *Landscape of complexity classes (final incarnation):* The class **co-NP** denotes the complement of the problem class **NP**. **co-NP** and **NP** have a nontrivial intersection that includes the class **P**, but also other, (probably) more difficult problems like FACTORING. Viewed as a pair of related problem classes, **NP** and **co-NP** form the basis of an entire hierarchy of problem classes that always come in pairs and expand outwards into **PSPACE**. This is called the *polynomial hierarchy* and is indicated by dots (to maintain readability).

convince others that this decomposition is correct. A *prime factorization* of $N \in \mathbb{N}$ looks like

prime factorization

$$N = F_0 \times F_1 \times \cdots \times F_{m-1} \quad \text{with} \quad F_0, \ldots, F_{m-1} \in \mathbb{N} \text{ prime.} \tag{11.2}$$

Moreover, the maximum number of factors m must obey $m \leq \log_2(N)$ (why?).

We can use multiplication to efficiently check whether the equality in Eq. (11.2) actually checks out. More remarkably, it is also possible to efficiently check that each proposed factor F_i is actually a prime number. This is courtesy of an algorithm discovered only in 2004 by the Indian computer scientists Agrawal, Kayal and Saxena [AKS04].

Theorem 11.1 (AKS primality test). There is a polynomial-time algorithm (TM) that decides the language $\textsc{Primes} = \{\llcorner F \lrcorner : F \text{ is a prime number}\} \subset \{0,1\}^*$.

Discussion and proof of this algorithm would go beyond the scope of this introductory lecture. For us, it is enough to know that it exists and works. Access to a (proposed) factorization (11.2) allows us to efficiently verify $\llcorner (N,k) \lrcorner \in \textsc{Factoring}$: simply check whether the proposed factorization is correct and (at least) one factor obeys $F_i \leq k$. The second step is also efficient, because there are at most $m \leq \log_2(N)$ factors that want to be checked. More formally:

Proposition 11.2 ((prime) FACTORING ∈ NP). A factorization (11.2) serves as a *short certificate* for $\llcorner(N,k)\lrcorner \in$ FACTORING. The verifier M checks whether (i) the factorization is correct and (ii) (at least) one factor obeys $F_i \leq k$.

Such an efficient verification procedure is characteristic of problems in **NP**. Other problems, like 3-SAT, TSP (traveling salesperson) and CLIQUE share this feature. But access to a factorization (11.2) allows us to do more than efficiently certify 'yes'-instances ($\llcorner(N,k)\lrcorner \in$ FACTORING). We can efficiently certify 'no'-instances ($\llcorner(N,k)\lrcorner \notin$ FACTORING) as well.

Proposition 11.3 ((prime) FACTORING is special). A factorization (11.2) also serves as a *short certificate* for $\llcorner(N,k)\lrcorner \notin$ FACTORING. The verifier M checks whether (i) the factorization is correct and (ii) none of the factors obeys $F_i \leq k$.

The first verification routine is identical to before. The second one is slightly different but also efficient, because we need to check at most $m \leq \log_2(N)$ different factors. This contrasts other **NP** problems. The traveling salesperson problem

$$\text{TSP} = \{\llcorner(D,k)\lrcorner : \text{ there is a TSP route for } D \text{ with} \leq k \text{ km}\}$$

is in **NP**, because we can efficiently certify 'yes' instances by providing a short route that gets by with at most k km. But, how would we certify 'no'-instances? Assuming proper encoding, $\llcorner(D,k)\lrcorner \notin$ TSP if and only if there is not a single route that achieves $\leq k$ kilometers. And checking the km-count of all $n!/2 = 2^{O(n^2)}$ possible routes is certainly not efficient.

A similar situation occurs for the posterchild of the problem class **NP**. For

$$\text{3-SAT} = \{\llcorner\varphi\lrcorner : \ \varphi \text{ is a 3-CNF that is satisfiable}\},$$

we can efficiently certify 'yes'-instances ($\llcorner\varphi\lrcorner \in$ 3-SAT) by providing a satisfiable assignment x such that $\varphi(x) = 1$. But, (again assuming proper encoding) 'no'-instances look like an entirely different beast:

$$\llcorner\varphi\lrcorner \notin \text{3-SAT} \quad \Leftrightarrow \quad \varphi(x) = 0 \text{ for all } x \in \{0,1\}^n.$$

How could we hope to efficiently convince others that the right hand side of this display is actually true for all 2^n possible inputs $x \in \{0,1\}^n$?

11.2 The problem class co-NP

We can capture the special nature of FACTORING by introducing a new complexity class. Recall that the *complement* of a language $A \subseteq \{0,1\}^*$ contains all strings not in the language, i.e.

$$\overline{A} = \{x : \ x \notin A\} \subseteq \{0,1\}^*.$$

We can use complements to formally define the class of all decision problems where 'no'-instances can be certified efficiently.

Definition 11.4 (co-NP). We say that a language $A \in \{0,1\}^*$ is in **co-NP** if its complement $\overline{A}$ is contained in **NP**. More formally: the problem class **co-NP**

$$\textbf{co-NP} = \left\{ A : \ \overline{A} \in \textbf{NP} \right\}.$$

Note that **co-NP** is *not* the complement of the problem class **NP**. In fact, both classes have a nontrivial intersection.

Lemma 11.5 The problem class **P** is in the intersection **NP** $\cap$ **co-NP**.

Proof. Suppose $A \in \mathbf{P}$, i.e. there exists a polynomial runtime TM M such that $M(x) = 1$ if and only if $x \in A$. Then, we can efficiently certify $x \in A$ by checking $M(x) \stackrel{?}{=} 1$. Hence, $A \in \mathbf{NP}$. But, we can also efficiently certify $x \notin A$ by checking $M(x) \stackrel{?}{=} 0$. Hence, $A \in$ **co-NP** as well. ■

The intersection **NP** $\cap$ **co-NP** also contains some languages for which we don't know any efficient algorithm.

Proposition 11.6 The language FACTORING introduced in Eq. (11.1) is contained in both **NP** and **co-NP**. FACTORING $\in$ **NP** $\cap$ **co-NP**

This is an immediate consequence of our analysis in Section 11.1. But the problem class **co-NP** is much larger and also contains interesting problems that behave very different from **NP** problems. The *tautology* problem in formal logic illustrates this discrepancy: tautology

$$\text{TAUTOLOGY} = \{ \llcorner \varphi \lrcorner : \ \varphi(x) = 1 \text{ for all inputs } x \}.$$

In words, tautology asks whether a given Boolean formula is completely trivial. A moment of reflection reveals that TAUTOLOGY behaves very differently from SAT. In particular, it is not enough to 'get lucky' and guess a single satisfying assignment. Instead, we must show that it is impossible (sic!) to produce $\varphi(\boldsymbol{x}) = 0$.

TAUTOLOGY is the **co-NP** analogue of SAT. And this analogy extends to polynomial-time (Karp) reductions.

Theorem 11.7 (TAUTOLOGY is co-NP complete). Let $A \in$ **co-NP** be a language. Then, there is a polynomial-time reduction of A to an instance of TAUTOLOGY. In formulas: $A \leq_p$ TAUTOLOGY and also TAUTOLOY $\in$ **co-NP**.

We won't do the proof here. It follows from adjusting the Cook-Levin reduction (see Lecture 8) from $\overline{A}$ (which is in **NP**) to SAT.

Most researchers believe that **NP** $\neq$ **co-NP**, although we do not know a formal proof. In fact, attempting to prove this conjecture is an ambitious goal. It is intimately related to the **P** vs. **NP**-conjecture, see Problem 11.17 below. Interestingly, we can also use our belief that **NP** $\neq$ **co-NP** to obtain a statement about the intrinsic hardness of FACTORING.

Theorem 11.8 Unless **NP** = **co-NP**, FACTORING cannot be **NP**-complete.

FACTORING is (probably) not 'that hard'

Proof. We will prove the contrapositive: If FACTORING is **NP**-complete, then **NP** ⊆ **co-NP** and **co-NP** ⊆ **NP**:
(i) **NP** ⊆ **co-NP**: By assumption, FACTORING is **NP**-complete. So, we can polynomial-time reduce *every* **NP**-language A to an instance of FACTORING ($A \leq_p$ FACTORING). But, we also know that this FACTORING-instance is in **co-NP**. In formulas: $A \in$ **NP** implies $A \leq_p$ FACTORING $\in$ **co-NP**.
(ii) **co-NP** ⊆ **NP**: Fix a language $B \in$ **co-NP**. Then, by Definition 11.4, $\overline{B} \in$ **NP** and, by assumption, $\overline{B} \leq_p$ FACTORING. Now, use the fact that polynomial-time (Karp) reductions are invariant under taking complements ($x \in B \Leftrightarrow f(x) \in$ FACTORING is logically equivalent to $x \in \overline{B} \Leftrightarrow f(x) \in \overline{\text{FACTORING}}$). Hence, $B \leq_p \overline{\text{FACTORING}}$. Finally, FACTORING $\in$ **co-NP** implies $\overline{\text{FACTORING}} \in$ **NP** (according to Definition 11.4). In formulas: $B \in$ **co-NP** implies $B \leq_p \overline{\text{FACTORING}} \in$ **NP**. ■

Theorem 11.8 hints at important differences between FACTORING and **NP**-complete problems like 3-SAT, TSP or CLIQUE. While we can polynomial-time reduce FACTORING to an instance of 3-SAT, the converse is unlikely to be possible. This indicates that *factoring numbers is probably not quite as difficult as solving the satisfiability problem*.

The belief that factoring is not quite as hard as other **NP**-problems is in line with the existence of Shor's algorithm. This is a quantum algorithm that promises to solve FACTORING in a runtime that is (roughly) cubic in input size. This efficient factoring algorithm, however, requires a quantum computer and cannot be carried out (efficiently) on conventional hardware. So, while Shor's algorithm doesn't quite prove FACTORING $\in$ **P** (because a quantum computer is fundamentally different from a TM), it might also suggest that the factoring problem may not be very hard after all.

existence of an efficient quantum algorithm for FACTORING

11.3 The polynomial hierarchy

We can slightly rewrite the definitions of **NP** and **co-NP** to make them appear more symmetric.

Definition 11.9 (NP, alternative definition). A language $A \in \{0,1\}^*$ is in **NP** if and only if there exists a polynomial runtime TM M and a polynomial q and such that

$$x \in A \quad \Leftrightarrow \quad \exists y \in \{0,1\}^{q(|x|)} \quad \text{such that} \quad M(x,y) = 1.$$

Definition 11.10 (co-NP, alternative definition). A language $A \in \{0,1\}^*$ is in **co-NP** if and only if there exist a polynomial runtime TM M and a polynomial q such that

$$x \in A \quad \Leftrightarrow \quad \forall z \in \{0,1\}^{q(|x|)} \quad \text{such that} \quad M(x,z) = 1.$$

Membership in **NP** is characterized by a single ∃-quantifier over polynomial-length bitstrings y. Similarly, membership in **co-NP** is characterized by a single

$\forall$-quantifier over polynomial-length bitstrings z. This makes sense, because we moved from **NP** to **co-NP** by negating (taking complements). And the negation of $\exists$ is $\forall$.

Note that this is beginning to look reminiscent of the 2-player Boolean formula games from Lecture 10. Those are fully quantified Boolean formulas where many $\exists$ and $\forall$-quantifiers alternate, e.g.

$$\exists x_0 \forall x_1 \exists x_2 \cdots \varphi(x_0, \ldots, x_{n-1}) \stackrel{?}{=} 1, \quad \text{or} \tag{11.3}$$

$$\forall x_0 \exists x_1 \forall x_2 \cdots \varphi(x_0, \ldots, x_{n-1}) \stackrel{?}{=} 1. \tag{11.4}$$

In order to interpolate between **NP** and **co-NP** problems on the one hand and fully quantified Boolean formulas on the other, we can introduce additional complexity classes where several $\exists$ and $\forall$ quantifiers alternate.

Definition 11.11 (the problem classes $\mathbf{\Sigma}_i^p$ for $i \geq 1$). For $i \geq 1$, we define the problem class $\mathbf{\Sigma}_i^p$ to contain all languages A for which there exists a polynomial time TM M and a polynomial q such that

$$x \in A \quad \Leftrightarrow \quad \begin{array}{l} \exists y_0 \in \{0,1\}^{q(|x|)} \, \forall y_1 \in \{0,1\}^{q(|x|)} \cdots Q_{i-1} y_{i-1} \in \{0,1\}^{q(|x|)} \\ M(x, y_0, y_1, \ldots, y_{i-1}) = 1, \end{array}$$

where Q_{i-1} is either $\forall$ or $\exists$ depending on the parity of i.

The essential requirement in the equation display starts with an $\exists$ quantifier. We can define related problem classes by starting with a $\forall$-quantifier instead.

Definition 11.12 (the problem classes $\mathbf{\Pi}_i^p$ for $i \geq 1$). For $i \geq 1$, we define the problem class $\mathbf{\Pi}_i^p$ to contain all languages A for which there exists a polynomial time TM M and a polynomial q such that

$$x \in A \quad \Leftrightarrow \quad \begin{array}{l} \forall z_0 \in \{0,1\}^{q(|x|)} \, \exists z_1 \in \{0,1\}^{q(|x|)} \cdots Q_{i-1} z_{i-1} \in \{0,1\}^{q(|x|)} \\ M(x, z_0, z_1, \ldots, z_{i-1}) = 1, \end{array}$$

where Q_{i-1} is either $\forall$ or $\exists$ depending on the parity of i.

These problem classes generalize **NP** (as re-introduced in Definition 11.9) and **co-NP** (as re-introduced in Definition 11.10). In particular,

$$\mathbf{\Sigma}_1^p = \mathbf{NP} \quad \text{and} \quad \mathbf{\Pi}_1^p = \mathbf{co\text{-}NP}. \tag{11.5}$$

Also, it makes sense to extend both definitions to $i = 0$ ('no quantifiers') by assigning

$$\mathbf{\Sigma}_0^p = \mathbf{\Pi}_0^p = \mathbf{P}.$$

The following inclusions are also immediate consequences of recursive definitions in terms of i quantified certificates that alternate between $\exists$ and $\forall$.

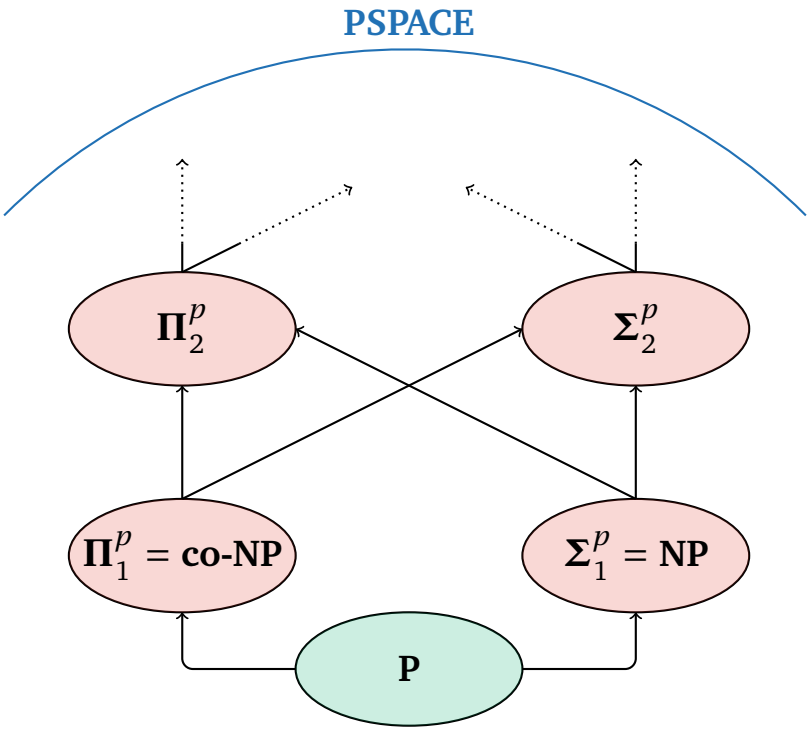


Figure 11.2 *Illustration of the polynomial hierarchy (PH).*

Lemma 11.13 The following inclusions are true for any $i \geq 0$:

$$\Sigma_i^p \subseteq \Sigma_{i+1}^p \quad \text{and} \quad \Sigma_i^p \subseteq \Pi_{i+1}^p, \quad \text{as well as,}$$
$$\Pi_i^p \subseteq \Pi_{i+1}^p \quad \text{and} \quad \Pi_i^p \subseteq \Sigma_{i+1}^p.$$

It looks as if these inclusions could go on forever. But there is a 'glass ceiling'. Membership in all of these problem classes can still be decided in polynomial space, see Lecture 10.

Proposition 11.14 For each $i \geq 1$, $\Sigma_i^p, \Pi_i^p \subseteq \mathbf{PSPACE}$.

Proof sketch. Re-use space to recursively check through all possible certificate choices $y_0, \ldots, y_{i-1} \in \{0,1\}^{q(|x|)}$ $(z_0, \ldots, z_{i-1})$ and i additional bits to keep track of the truth of $\exists$- and $\forall$-quantifiers. ■

As i increases, the problem classes Σ_i^p and Π_i^p seem to get bigger and encompass more difficult decision problems. We subsume *all* these classes in the *polynomial hierarchy* (**PH**):

polynomial hierarchy (**PH**)

$$\mathbf{PH} = \bigcup_{i \geq 0} \Sigma_i^p = \bigcup_{i \geq 0} \Pi_i^p.$$

Proposition 11.14 implies that the entire polynomial hierarchy is contained in **PSPACE**. The interdependencies between constituents of **PH** are illustrated in

Figure 11.2. We believe that $\mathbf{P} \neq \mathbf{NP}$ and $\mathbf{NP} \neq \mathbf{co\text{-}NP}$. Appealing generalizations of these conjectures are

collapse of the **PH**

(i) for each $i \geq 1$, $\mathbf{\Sigma}_i^p$ is strictly contained in $\mathbf{\Sigma}_{i+1}^p$ ($\mathbf{\Sigma}_i^p \subset \mathbf{\Sigma}_{i+1}^p$); and
(ii) for each $i \geq 1$, $\mathbf{\Sigma}_i^p$ and $\mathbf{\Pi}_i^p$ are distinct ($\mathbf{\Sigma}_i^p \neq \mathbf{\Pi}_i^p$).

These assumptions are often summarized by a single conjecture: *the polynomial hierarchy does not collapse*.

Theorem 11.15 If $\mathbf{\Sigma}_i^p = \mathbf{\Pi}_i^p$ for some $i \geq 1$, then $\mathbf{PH} = \mathbf{\Sigma}_i^p$. I.e. the polynomial hierarchy collapses to the i-th level.

We leave the proof as an exercise (see problem-section below). It is widely believed that the polynomial hierarchy does not collapse to any level i. Many interesting complexity-theoretic arguments follow from such assumptions. The following one is an immediate consequence of Theorem 11.8 and Eq. (11.5).

Corollary 11.16 If FACTORING is **NP**-complete, then the polynomial hierarchy collapses to the *first* level ($i = 1$). In formulas: $\mathbf{PH} = \mathbf{\Sigma}_1^p = \mathbf{NP}$.

We will discuss other interesting consequences in future lectures about circuits. In fact, low-order constituents of the **PH** encompass many important problems in circuit and Boolean function design. One of them is *circuit minimization*: find the shortest logical circuit that implements a given Boolean function. But more on that later.

Problems

Problem 11.17 (co-NP $\neq$ NP vs. P = NP).

1 Prove the following statement: if **NP** = **P**, then **co-NP** = **NP**.
2 Which of the following statements is a consequence of this claim?
    1 **co-NP** = **NP** implies **NP** = **P**;
    2 **co-NP** $\neq$ **NP** implies **NP** $\neq$ **P**;
    3 **NP** $\neq$ **P** implies **co-NP** $\neq$ **NP**;
    4 all of the above are true.

Problem 11.18 (Proof of Theorem 11.15). Show that $\mathbf{\Sigma}_i^p = \mathbf{\Pi}_i^p$ for some $i \geq 1$ implies $\mathbf{PH} = \mathbf{\Sigma}_i^p$.
Hint: It is enough to prove that $\mathbf{\Sigma}_i^p = \mathbf{\Pi}_i^p$ implies $\mathbf{\Sigma}_j^p = \mathbf{\Sigma}_{j-1}^p$ for every $j > 1$.

12. Circuits

Date: 11 January 2024

Agenda

Up to now, the central object of our study has been the Turing machine (TM). TMs are an abstract model for *algorithms* and we used this model to make statements about computing problems. The cost parameters we used to quantify such claims have been *runtime* and (memory) *space*. We used these to group decision problems (languages) into different complexity classes.

Today we introduce another model of computation: (logical) *circuits*. They are a simplified model of the silicon chip – the basic hardware device that makes up modern computers. Circuits can be used to compute any logical function and come with their own natural cost parameters: *size* and also *depth*.

The circuit model of computation provides an alternative perspective on computational complexity. And it is interesting to compare the circuit model of computation to the TM model of computation. This will be the content of the final three lectures. Today, we introduce circuits from a theoretical point of view. We will show that it is possible to construct circuits that compute arbitrary logical functions. In fact, we will prove that they can even simulate entire TM computations with only a logarithmic overhead in computation complexity. This provides a high-level argument for the existence of efficient special-purpose hardware, such as calculators, as well as efficient general-purpose hardware such as *processing units*.

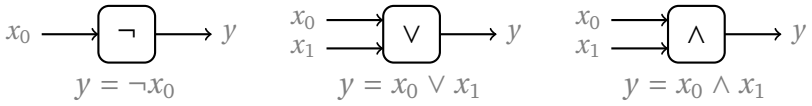
Agenda:

1. (Logical) circuits
2. Circuit size and depth
3. Representing logical functions as circuits
4. Representing TM computations via circuits
5. Circuits for universal TMs

12.1 (Logical) circuits

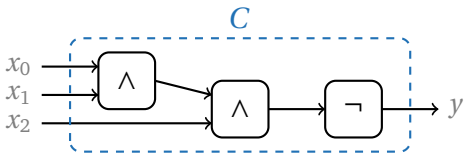
The elementary building blocks of a logical circuit are ¬ (NOT), ∨ (OR) and ∧ (AND). Pictorially, we represent each of these logical operations by a square box[1]:

elementary logical gates



Each square box is called a (logical) *gate*. The lines entering on the left denote input bits (also called fan-in), while the lines exiting on the right denote output bits (also called fan-out). All three elementary logical gates have a single output bit. The 'NOT'-gate also has a single input bit, while the 'OR'-gate and 'AND'-gate have two each.

We can now use these gates as elementary building blocks to build more complicated objects that map n input bits (on the very left) to m output bits (on the very right). These more complicated configurations are called (logical) *circuits* Here is a simple example circuit C involving two ∧-gates and one ¬-gate:



(12.1)

This circuit has $n = 3$ input bits (x_0, x_1, x_2) and $m = 1$ output bit (y). The wires (arrows) tell us how to propagate the three input bits through the circuit. Whenever one or two of them enter a gate, we apply the associated logical functionality. This produces a new (intermediate) bit that we treat as a new input for gates that are located deeper within the circuit. At the very right, only one arrow exits this particular circuit. It indicates the single output bit (y). The total circuit C (highlighted by a dashed blue box) computes the following logical function:

$$y = C(x_0, x_1, x_2) = \neg\left((x_0 \wedge x_1) \wedge x_2\right) = \begin{cases} 0 & \text{if } x_0 = x_1 = x_2 = 1, \\ 1 & \text{else.} \end{cases}$$

We see that circuits provide a graphical set of instructions on how to execute and combine elementary logical operations.

Definition 12.1 (Circuit, informal). A (logical) circuit connects n input bits with m output bits via a network of directed wires that connect different elementary logical operations $(\wedge, \vee, \neg)$. The result is a logical function $C : \{0,1\}^n \to \{0,1\}^m$.

circuits are instructions for logical operations

[1] This notation convention differs from the symbols used in electrical engineering and hardware design.

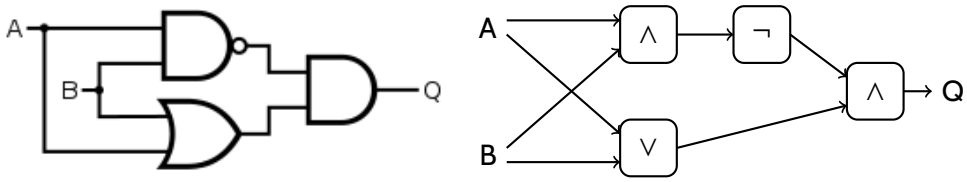


Figure 12.1 *Representation of XOR as logical circuit:* **(left)** standard circuit diagram involving NAND, OR and AND, taken from Wikipedia. **(right)** Reformulation as a logical circuit. It looks like a directed graph. Two nodes – A and B – have no incoming edges. These are the *input nodes*. One node – Q – has no outgoing edge. This is the *output node*. Nodes in-between always have one outgoing edge and either one ($\neg$) or two ($\vee$, $\wedge$) incoming edges, respectively.

From a high-level perspective, circuits describe a directed flow of information processing. Formally, each such network can be described by a *graph*, i.e. a set of nodes (or vertices) that are connected by edges. Each elementary gate corresponds to a node and the edges tell us which elementary gates are connected. Importantly, information flow within a circuit only goes in one direction (the one indicated by arrow-heads). So, the graph is actually a *directed graph*, because edges always have a pre-specified orientation. Input and output bits also correspond to nodes in the directed graph. Input bits are described by graph nodes with no incoming edges ('sources'), while output bits are graph nodes with no outgoing edges ('sinks'). We always try to plot/draw circuit graphs such that the input nodes appear on the very left, while the output nodes appear on the very right. See Figure 12.1 for a visual illustration. With this convention, information always propagates from left to right. Such a directed flow of information is an important characteristic of circuits. But it only makes sense if the underlying graph does not back-propagate information. In other words: the underlying graph should not have loops or *cycles*. Graphs without cycles are called *acyclic*. (Note that the absence of cycles is necessary to clearly identify inputs and outputs within the circuit.) We are now ready to provide a mathematically rigorous definition for (logical) circuits.

circuits are directed, acyclic graphs

Definition 12.2 (Circuit, formal). A *(logical) circuit* C with n inputs and m outputs is a directed acyclic graph. There are n *input nodes* with no incoming edges and m output nodes with no outgoing edges. Nodes in-between are called *gates* and correspond to either $\vee$ (OR), $\wedge$ (AND) or $\neg$ (NOT). Viewing the graph as a set of logical instructions produces a logical function $C : \{0,1\}^n \to \{0,1\}^m$.

12.2 Circuit size and circuit depth

Definition 12.2 provides a rigorous connection between circuits and graphs. This connection can be used to identify a variety of meaningful summary parameters. For instance, the number of input nodes n simply corresponds to the *input length*. Likewise, the number of output nodes m is the *output length*. The

input and output length

following two summary parameters tell us something about how complicated a circuit actually is.

Definition 12.3 (circuit size and circuit depth). The *size* $s(C) \in \mathbb{N}$ of a circuit C is the total number of gates (i.e. the number of nodes that do not correspond to either input or output nodes). The *depth* $d(C) \in \mathbb{N}$ of a circuit is the length of the longest directed path from an input node to an output node (not counting edges adjacent to input and output nodes).

circuit size and circuit depth

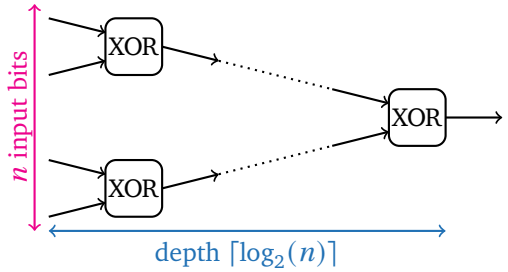
Note that the concept of depth only makes sense if the underlying graph does not contain any cycles. This is one of the reasons why we require acyclic graphs in Definition 12.2. For small and/or highly structured circuits, it is typically easy to determine size and depth by just looking at the circuit. Take Eq. (12.1) as a first example. This circuit has size $s(C) = 3$ and depth $d(C) = 3$. The circuit in Figure 12.1 has size $s(\text{XOR}) = 4$ and depth $d(\text{XOR}) = 3$.

The *size* of a circuit counts the total number of elementary logical operations we need to carry out. This is reminiscent of TM runtime that counts the total number of elementary steps a TM must execute to arrive at a desired solution.

circuit size ≈ TM runtime

Circuit depth is a bit more complicated by comparison, because it depends on the information flow within the circuit. The following example shows that size and depth can deviate by an exponential factor.

Example 12.4 (Size and depth of a circuit that computes parity of sums). Consider a circuit $C_{\text{ParityOfSum}} : \{0,1\}^n \to \{0,1\}$ that corresponds to a binary tree graph with n leaves (input nodes) and $m = 1$ root (output node):



(12.2)

Each gate corresponds to a XOR-operation. There are $n-1$ such gates in total, but the depth of the tree graph is only $\lceil \log_2(n) \rceil$. If we use the decomposition from Figure 12.1 to further subdivide each XOR gate into elementary gates ($\vee$, $\wedge$ and $\neg$), we obtain

$$\begin{aligned} s\left(C_{\text{ParityOfSum}}\right) &= s\left(\text{XOR}\right)(n-1) = 4(n-1), \\ d\left(C_{\text{ParityOfSum}}\right) &= d\left(\text{XOR}\right)\lceil \log_2(n) \rceil = 3\lceil \log_2(n) \rceil. \end{aligned}$$

We see that the depth of this circuit is exponentially smaller than its size! What is more, the underlying circuit actually computes an old friend of ours. Recall that XOR computes the sum of two bits modulo 2: $\text{XOR}(a,b) = a \oplus b = a + b \mod 2$. Hence,

$$\begin{aligned} C_{\text{ParityOfSum}}\left(x_0, \ldots, x_{n-1}\right) &= x_0 \oplus x_1 \oplus \cdots \oplus x_{n-1} \\ &= \text{parity}\left(x_0 + \cdots + x_{n-1}\right) \end{aligned}$$

and we see that this circuit actually allows us to decide whether the number of 1s in a given length-n bitstring $x = x_0 \cdots x_{n-1}$ is even ($C_{\text{ParityOfSum}}(x) = 0$) or odd ($C_{\text{ParityOfSum}}(x) = 1$). ▪

The above example highlights that circuit size and circuit depth can differ vastly from each other. Circuit size can be thought as the time taken by a sequential machine to compute a given function. Circuit depth, in contrast, measures the 'parallelized' time required to finish the same job. Indeed, the underlying graph structure ensures that elementary gates within the same (vertical) layer can be executed concurrently. And the circuit depth counts the total number of such layers from beginning to end.

circuit depth ≈ parallelized runtime

The potential for *parallelization* is the first of several concepts that are naturally captured by circuits (via the circuit depth), but actually take quite a lot of care to appropriately capture in the TM model of computation.

12.3 Representing logical functions as circuits

Example 12.4 is instructive in more ways than one. The circuit presented in Eq. (12.2) provides a very efficient way to compute a simple Boolean function (recall that Boolean functions have n input bits and a single output bit):

$$f_{\text{ParityOfSum}}(x_0, \ldots, x_{n-1}) = \begin{cases} 0 & \text{if } x_0 + \cdots + x_{n-1} \text{ is even,} \\ 1 & \text{if } x_0 + \cdots + x_{n-1} \text{ is odd.} \end{cases}$$

We can associate this Boolean function with the language (decision problem)

$$\text{ParityOfSum} = \left\{ x \in \{0,1\}^* : \ \text{parity}\left(\sum_i x_i\right) \right\} \subset \{0,1\}^* .$$

In fact, we have already analyzed this particular language in the first part of the course. ParityOfSum is a *regular language*, meaning that we can always decide membership with a deterministic finite state automaton (DFA). Regular languages capture some of the easiest decision problems conceivable, because the runtime of a DFA is always equal to input length n. The circuit depicted in Eq. (12.2) also captures this feature, because it contains exactly $(n-1)$ XOR-gates. But it tells us something else. Executing these simple two-bit operations in parallel would allow us to compress runtime down to only $3\lceil \log_2(n) \rceil$ computational steps – an exponential improvement over the (sequential) working of the DFA.

We see that (logical) circuits and (logical) functions are intimately connected. In fact, every circuit C computes a logical function $f_C : \{0,1\}^n \to \{0,1\}^m$. The other direction is also true, and more interesting. We can represent *any* logical function by circuits.

circuits compute logical functions

Theorem 12.5 (universality of circuits). Let $f : \{0,1\}^n \to \{0,1\}^m$ be a logical function with n input bits and m output bits. Then, we can construct a logical circuit C_f comprised of only $\neg$-, $\vee$- and $\wedge$-gates that implements this

logical functions can always be represented as circuits

function. Circuit size and depth obey

$$s\left(C_f\right) \leq mn2^{n+1} - 1 = O\left(mn2^n\right) \qquad \text{(circuit size)},$$
$$d\left(C_f\right) \leq n + \lceil \log_2(n) \rceil + 1 = O(n) \qquad \text{(circuit depth)}.$$

This general statement highlights that circuits implementing logical functions may require exponentially many gates (in input length n) to implement. But their depth is at most linear in input length.

We can prove this claim by remembering a powerful statement about the universality of elementary logical operations $(\neg, \wedge, \vee)$. Recall that a function $\varphi : \{0,1\}^n \to \{0,1\}$ is in k-CNF if it can be written as an AND of Or's of potentially negated variables. The OR-expressions are called clauses and can contain (at most) k variables or their negations. The following result from Lecture 8 reminds us that we can convert *any* logical function into CNF.

Theorem 12.6 (Theorem 8.8 from Lecture 8). Let $f : \{0,1\}^n \to \{0,1\}$ be a Boolean function on n bits. Then, we can construct a n-CNF φ that contains (at most) 2^n and obeys $\varphi(x) = f(x)$ for all $x \in \{0,1\}^n$.

Such a conversion into n-CNF is extremely useful when one attempts to design circuits with short circuit depth. Much like the circuit from Example 12.4, CNFs have a lot of potential for parallelization.

Lemma 12.7 Suppose that $\varphi : \{0,1\}^n \to \{0,1\}$ is a Boolean function in k-CNF that contains (at most) l clauses. Then, we can represent φ by a circuit C_φ with size $s\left(C_\varphi\right) \leq 2kl - 1 = O(kl)$ and depth $d\left(C_\varphi\right) \leq \lceil \log_2(k) \rceil + \lceil \log_2(l) \rceil + 1 = O\left(\log(kl)\right)$.

We leave the proof as an exercise. Together, Theorem 12.6 and allow us to readily prove the universality claim about circuits.

Proof of Theorem 12.5. We can decompose the logical function $f : \{0,1\}^n \to \{0,1\}^m$ into m disjoint Boolean functions $f_i : \{0,1\}^n \to \{0,1\}$ – one for each output variable. Theorem 12.6 asserts us that we can convert each of these Boolean functions into a CNF φ_i with clause size $k_i \leq n$ and a total number of (at most) $l_i \leq 2^n$ such clauses. Subsequently, we can use Lemma 12.7 to convert each CNF φ_i into a circuit C_{φ_i} with n input bits, one output bit, size $s_i = s\left(C_{\varphi_i}\right) \leq 2l_i k_i - 1 = n2^{n+1} - 1 = O(n2^n)$ and depth $d_i = d\left(C_{\varphi_i}\right) \leq \lceil \log_2(l_i) \rceil + \lceil \log_2(k_i) \rceil + 1 \leq n + \lceil \log_2(n) \rceil + 1 = O(n)$.

Now, it's time to recall that we are actually dealing with m circuits – one for each output bit. But, they can all be computed completely independently from each other! Hence, we get a total circuit size $s(C_f) = \sum_{i=1}^m s_i \leq mn2^{n+1} - 1 = O(mn2^n)$ and total circuit depth $d(C_f) = \max_i d_i \leq n + \lceil \log_2(n) \rceil + 1 = O(n)$. ■

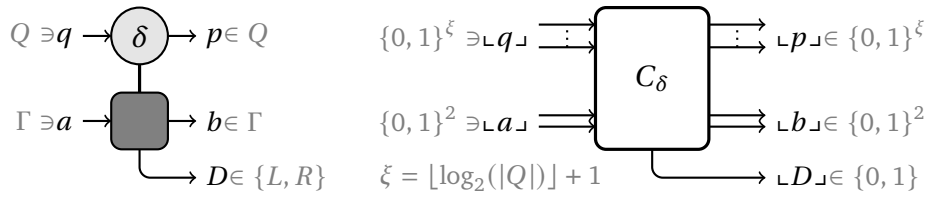


Figure 12.2 *TM transition function as circuit:* **(left)** a TM transition function δ takes two inputs – a state q and a tape symbol a – and produces three outputs – another state p, another tape symbol b, as well as a direction D for the next tape move. If we encode all the symbols involved into bitstrings, the transition function becomes a logical function that maps bitstrings onto bitstrings. **(right)** We can represent the (encoded) transition function δ by an equivalent circuit C_δ. The size of this circuit is guaranteed to be constant ($O(1)$).

12.4 Representing TM computations via circuits

Theorem 12.5 states that we can represent any logical function as a circuit. This provides an interesting connection between formal logic (Boolean functions) and hardware design (circuits). We can always construct a silicon chip that implements a desired logical functionality. But can we go one step further and establish a similar connection between algorithms (represented as TMs) and hardware design (represented by circuits as well)?

Recall that a TM is formally described by a 7-tuple

$$M = \left(Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}}\right).$$

Without loss, we can furthermore assume that the state alphabet is binary ($\Sigma = \{0,1\}$) and the tape alphabet only contains four symbols: $\Gamma = \{0, 1, \square, \#\}$. As usual, '$\square$' denotes empty tape space and # is an additional roadblock symbol that will become useful later on. What matters now already is that we can encode all four tape symbols into two bits (e.g. by specifying $0 \leftrightarrow 00$, $\square \leftrightarrow 01$, $\# \leftrightarrow 10$, $1 \leftrightarrow 11$). Likewise, we can represent all possible internal states $q \in Q$ by bitstrings $\llcorner q \lrcorner$ of length $\xi = \lfloor \log_2(|Q|) \rfloor + 1$. It will also make our life easier later on, if we choose a sensible labeling convention where $q_0 = \llcorner 0 \lrcorner = 0 \cdots 0$ and $q_{\text{accept}} = \llcorner 2^\xi \lrcorner = 1 \cdots 1$. Finally, we can use one additional bit $\llcorner D \lrcorner$ to indicate tape movements $D \in \{L, R\}$: $\llcorner L \lrcorner = 0$ (going left) and $\llcorner R \lrcorner = 1$ (going right). To summarize, we have replaced all static constituents of a TM by bitstrings of appropriate size. Such a bit encoding of static parts also has profound implications on the dynamical aspects of the TM. We can now replace the transition function

$$\begin{aligned} \delta : Q \times \Gamma &\to Q \times \Gamma \times \{L, R\}, \\ (q, a) &\mapsto (p, b, D) \end{aligned}$$

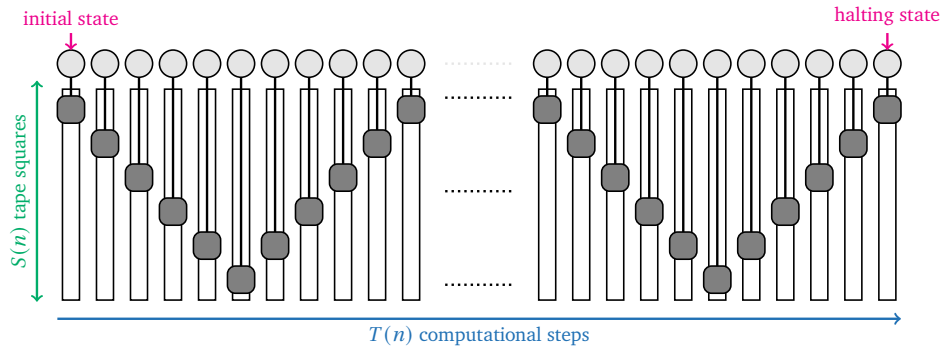


Figure 12.3 *Computational trajectory of zig-zag TM:* a section of admissible tape space is specified (green arrow on the left), the TMs tape is initially positioned on top of the very first tape square. It then sweeps downwards, and modifies tape squares one step at a time. Once it hits the last admissible tape space, the tape head movement reverts and goes back up. This zig-zag movement continues for a total of $T(n)$ elementary steps (blue arrow on the bottom). All the while, the internal states may change, but the TM is guaranteed to end up in a halting state (q_{accept} or q_{reject}) after the final sweep is concluded.

by a logical function that maps bitstrings onto bitstrings:

$$f_\delta : \{0,1\}^\xi \times \{0,1\}^2 \to \{0,1\}^\xi \times \{0,1\}^2 \times \{0,1\}\,,$$
$$(\llcorner q \lrcorner, \llcorner a \lrcorner) \mapsto (\llcorner p \lrcorner, \llcorner b \lrcorner, \llcorner D \lrcorner).$$

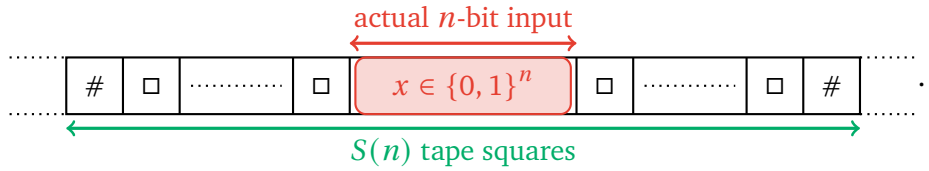
In total, this logical function features $n' = \xi + 2 = O(1)$ and $m' = \xi + 3 = O(1)$ output bits (recall that $\xi = \lfloor \log_2(|Q|) \rfloor + 1$). So, according to Theorem 12.5, we can represent f_δ by a logical circuit C_δ. The size of this circuit only depends on the constant number of TM states $|Q|$:

$$s\left(C_\delta\right) \leq m' n' 2^{n'+1} = O\left(|Q| \log^2(|Q|)\right) = O(1). \tag{12.3}$$

Our transformation from transition function δ to a circuit C_δ with constant size is visualized in Figure 12.2. It is an important first step to efficiently capture associated TM computations by a single logical circuit.

In fact, we already have all the pieces in place to construct circuits that represent TMs with simple and predictable head-movements. A *zig-zag TM* M processes length-n inputs x by repeatedly zig-zagging across a fixed section of $S(n)$ tape squares. Initially, the tape content looks like

zig-zag TM



Here, we use the additional alphabet symbol # (think roadblock) to delineate a finite section of the tape to which the TM computation is confined. The actual

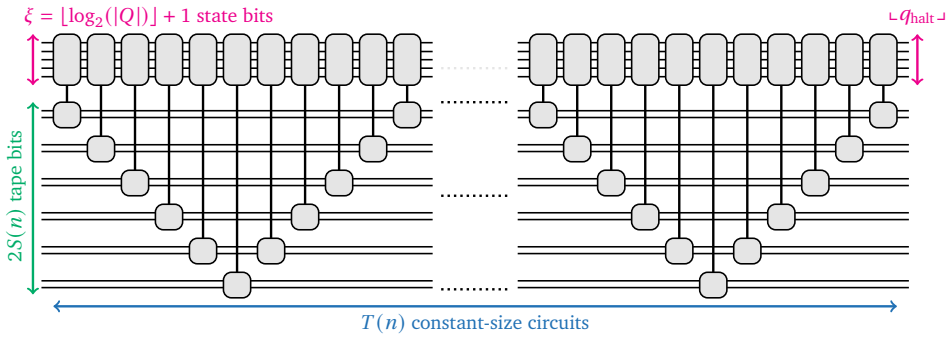


Figure 12.4 *Circuit that represents the entire computation of a zig-zag TM:* as visualized in Figure 12.3, zig-zag TMs have very predictable head movements. This allows us to replace the entire TM computation by a single circuit. The first $\xi = \lfloor \log_2(|Q|) \rfloor + 1$ bits are used to keep track of the internal TM state throughout the computation. The remaining $2S(n)$ bits are used to encode tape symbols. Subsequently, each TM step can be represented by applying the transition function circuit C_δ to the relevant subcollection of bits. The resulting circuit reflects the zig-zag motion of the original TM computation and comes with only a constant overhead in circuit size.

input $x \in \{0,1\}^n$ (red) is written into the center of this segment followed by empty tape space ($\square$) towards the left and right. A zig-zag TM starts with its tape head on the #-symbol towards the very left. It then moves leftwards across all $S(n)$ designated tape squares and uses its transition function to (potentially) change tape content along the way. Once it hits the #-symbol on the very right, it reverses its direction and moves back again. This zig-zagging motion of the tape head is visualized in Figure 12.3 and continues for a total of $T(n)$ computational steps. Once the last zig-zag is concluded, the zig-zag TM enters a halting state q_{halt} that either accepts or rejects the input string x:

$$M(x) = \begin{cases} 0 & \text{if } q_{\text{halt}} = q_{\text{reject}}, \\ 1 & \text{if } q_{\text{halt}} = q_{\text{accept}}. \end{cases}$$

We have already encountered zig-zag TMs throughout the course of this lecture. The TM that checks for palindrome structure (i.e. whether a bitstring x equals its own reverse x^{R}) in Lecture 3 is one example. The oblivious TM from an Exercise Sheet is another one – but more on that later. For now it suffices to notice that the tape movements of zig-zag TMs are extremely regular and independent from the content of the actual input string x. This allows us to represent the entire computational trajectory by a circuit. To achieve this, we simply use the circuit representation C_δ of the transition function δ from Figure 12.2. This necessitates an additional number of $\xi = \lfloor \log_2(|Q|) \rfloor + 1$ wires to keep track of the internal TM states. We also represent each of the $S(n)$ designated tape squares by two bits that store its tape symbol ($0 \leftrightarrow 00$, $\square \leftrightarrow 01$, $\# \leftrightarrow 10$, $1 \leftrightarrow 11$). Now, we use the fact that the tape head

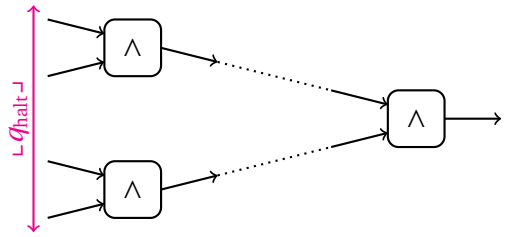
circuit for zig-zag TMs

movements of a zig-zag TM is so simple and predictable. We can simulate them by simply concatenating C_δs that act on the relevant bit pairs. Visualized in Figure 12.4 the resulting circuit $C_{M,n}$ closely resembles the computational trajectory of the zig-zag TM M in question.

The entire circuit has $2S(n) + \xi$ input bits, but most of them are set to 0 (recall that $\square$ is encoded by 00 and q_0 is encoded by $0 \cdots 0$). Only the $2n$ input bits at the center really matter, because they encode the actual input x. The circuit itself is composed of $T(n)$ variants of C_δ affecting different wires. This produces a total size of

$$s\left(C_{M,n}\right) = s\left(C_\delta\right) \times T(n) = \text{const} \times T(n),$$

where the constant stems from Eq. (12.3) and only depends on $|Q|$. A quick look at the circuit geometry displayed in Figure 12.4 reveals that the circuit depth must be roughly comparable to the circuit size – there simply isn't much room for parallelization. Now, the only remaining thing is how to extract a single output bit. The circuit should produce 0 if the TM M rejects the input ($q_{\text{halt}} = q_{\text{reject}}$) and 1 if the TM M accepts the input ($q_{\text{halt}} = q_{\text{accept}}$). But, this is easy. Suppose that have labeled our internal TM states such that $\llcorner q_{\text{accept}} \lrcorner = 1 \cdots 1$ ($\llcorner q_{\text{reject}} \lrcorner$ can be any other bitstring). Then, we can simply compute the AND of all $\xi = \lfloor \log_2(|Q|) \rfloor + 1$ final state bits:



This final circuit has size $\xi = \lfloor \log_2(|Q|) \rfloor + 1 = O(1)$ and produces a single output bit with the desired functionality. The bits contained in all other remaining wires don't matter and can be forgotten. Putting everything together, we obtain the following result that deserves a proposition-environment.

circuit for zig-zag TM

Proposition 12.8 (circuit representation of zig-zag TMs). Let M be a zig-zag TM (i.e. a TM whose tape head sweeps back and forth across a fixed segment of tape space) that accepts/rejects length-n inputs in runtime $T(n)$ and space $S(n)$. Then, for each input length n, we can represent the entire TM computation by a (Boolean) circuit $C_{M,n}$ with $n' = (2S(n) + c_1) = O(S(n))$ input bits, $m' = 1$ output bit, and size $s\left(C_{M,n}\right) = c_2 \times T(n) + c_3 = O(T(n))$ such that $C_{M,n}(x) = M(x)$ for all $x \in \{0,1\}^n$. The constants c_1, c_2, c_3 only depend on the number of internal TM states $|Q|$.

Note that the overhead in terms of input length and circuit size is only constant! It is also worthwhile to emphasize that we proved this statement by writing down a circuit with the desired properties. The proof is constructive, in the sense that we could actually build the circuit for any given zig-zag TM of

interest. But an overarching question remains: are zig-zag TMs useful to begin with? The following statements answers this question.

zig-zag TMs can efficiently simulate every other TM

Proposition 12.9 (universality of zig-zag TMs). Let M be an arbitrary TM with runtime $T(n) \geq n$. Then, we can simulate M by a zig-zag TM $\tilde{M}$ with runtime $\tilde{T}(n) = O\left(T(n)^2\right)$ and space $\tilde{S}(n) = O\left(T(n)\right)$.

We leave the proof as an instructive exercise in TM simulation. A more careful analysis based on oblivious TMs actually improves the runtime overhead from $O(T(n)^2)$ (quadratic) to only $O\left(T(n)\log(T(n))\right)$ (logarithmic), see [AB09, Proof sketch of Theorem 6.7]. A combination of Proposition 12.8 and Proposition 12.9 then yields the main result of this lecture:

circuit for arbitrary TMs

Theorem 12.10 (representing TMs as circuits). Let M be an arbitrary TM with runtime $T(n) \geq n$. For each input length n, we can implement M as a circuit $C_{M,n}$ with $n' = O\left(T(n)\right)$ input bits (most of them initialized to zero), $m' = 1$ output bit (0 for reject, 1 for accept) and size $s\left(C_{M,n}\right) = O\left(T(n)^2\right)$. This circuit obeys $C_{M,n}(x) = M(x)$ for all $x \in \{0,1\}^n$.

As pointed out above, the quadratic overhead in circuit size can actually be improved to only a logarithmic overhead:

$$s\left(C_{M,n}\right) = O\left(T(n)\log(T(n))\right).$$

This showcases that the transition from TMs – which are a simplified model of algorithms or programs – to circuits – which are a simplified model of silicon chips – can always be executed rather efficiently. Viewed from this angle, Theorem 12.10 provides a high-level argument for the existence of hardware that executes special-purpose computations (aka a single TM) very efficiently. A *calculator* comes to mind in this context.

circuits require fixed input length

Warning 12.11 Circuits can only process inputs of fixed length n (*nonuniform computation*). This is in stark contrast to TMs which can process inputs of arbitrary length (*uniform computation*). Hence, one TM M gives rise to an entire family of circuits $\left\{C_{M,n}\right\}_{n\in\mathbb{N}}$ – one circuit for each input length n. ▪

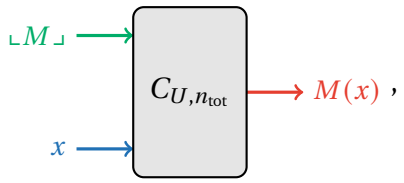
12.5 Circuits for universal TMs

But, we can do even better. Recall the concept of a *universal Turing machine* U from Lecture 5. Such a TM takes two inputs: (i) a bit encoding $\llcorner M \lrcorner$ of another TM M and (ii) (a bit encoding of) an input x of interest. It subsequently simulates the TM M on input x:

$$U\left(\llcorner M \lrcorner, x\right) = M(x).$$

What is more, one can show that the runtime overhead is at most logarithmic. I.e. if M has runtime $T(n)$, then U has runtime $O(T(n)\log(T(n)))$.

In turn, we can use the construction from Proposition 12.8 (refined to incur only a logarithmic overhead) to convert U into a circuit $C_{U,n_{\text{tot}}}$ of size $s\left(C_U\right) = O\left(T(n)\log^2(T(n))\right)$. Doing so produces a circuit that depends on the (maximum) total input size $n_{\text{tot}} = |(\llcorner M \lrcorner, x)|$. It takes two inputs – a 'program description' $\llcorner M \lrcorner$ and an actual input x – and computes the result $M(x)$. This can be visualized as



and actually describes a hardware device that is programmable. This can be viewed as an abstract model for an actual processor. Again, the overhead required to simulate many different programs (TMs) is only poly-logarithmic: a universal circuit comprised of $O\left(T_{\text{max}}\log^2(T_{\text{max}})\right)$ elementary gates can execute *any* TM computation whose runtime is bounded by a threshold value T_{max} that depends on n_{tot}.

circuits that are programmable

Problems

Problem 12.12 (Proof of Lemma 12.7). Prove the following claim: Suppose that $\varphi : \{0,1\}^n \to \{0,1\}$ is a Boolean function in k-CNF that contains (at most) l clauses. Then, we can represent φ by a circuit C_φ with size (at most) $2lk - 1 = O(lk)$ and depth (at most) $\lceil \log_2(k) \rceil + \lceil \log_2(l) \rceil + 1 = O\left(\log(lk)\right)$.

Problem 12.13 (Proof of Proposition 12.9). Prove the following claim: *any* runtime-$T(n)$ TM by a zig-zag TM with runtime $\tilde{T}(n) = O\left(T(n)^2\right)$ and space $\tilde{S}(n) = O\left(T(n)\right)$.

13. Circuit size-bounded computations

Date: 18 January 2024

Agenda

Last time, we introduced circuits as an alternative model of computation. Today, we continue our study of computational complexity from a circuit perspective. We define the complexity class **PSIZE** which encompasses all decision problems that can be solved by circuit families of polynomial size. This new problem class includes **P**, the class of problems that can be solved by polynomial runtime Turing machines (TMs). Remarkably, the converse relation is not true (**PSIZE** $\not\subseteq$ **P**): **PSIZE** even contains problems that are uncomputable in the TM model! There is an interesting story behind this discrepancy: short circuits correspond to polynomial-time TMs that are empowered by access to external advice. Not subject to any computational restrictions, such advice strings can be powerful game changers. TMs with advice allow us, in particular, to model nontraditional problem solving strategies, like various *machine learning* models. However, we will see that there are also limitations: the Karp-Lipton Theorem shows that, unless the polynomial hierarchy collapses, polynomial runtime TMs empowered by advice are not able to solve **NP**-complete problems (**NP** $\not\subseteq$ **PSIZE**). This thwarts hopes for devising efficient circuit-based solutions for important, but hard problems, like 3-SAT.

Agenda:

1. (Circuit) size-bounded computations
2. TMs that take advice
3. Karp-Lipton Theorem

13.1 (Circuit) size-bounded computations

In this course, we model computational problems by decision problems (yes/no-questions) or, equivalently, *languages*:

$$A \subseteq \{0,1\}^* .$$

TMs are a uniform model of computation

To decide membership, we usually feed candidate bitstrings x into a Turing machine (TM). A TM M decides/recognizes the language A if $x \in A \Leftrightarrow M(x) = 1$ and $x \notin A \Leftrightarrow M(x) = 0$ for all possible inputs $x \in \{0,1\}^*$. A single, fixed TM can process inputs of arbitrary length $n = |x|$. This allows us to also capture the language A by a single TM M that works for all input sizes. We say that TMs are a *uniform model of computation*.

circuits are a non-uniform model of computation

Alternatively, we can also use *circuits* to decide whether $x \in A$. But in this case, we may need a separate circuit for each input length n. This results in an entire *circuit family* $\{C_n\}_{n\in\mathbb{N}}$ – one for each input size n – decides membership for language A. Each circuit family member C_n is a (Boolean) circuit that has exactly n input bits and one output bit ($m = 1$). We say that a circuit family $\{C_n\}_{n\in\mathbb{N}}$ decides/recognizes the language A if

$$x \in \{0,1\}^n \quad \text{is contained in } A \text{ if and only if} \quad C_n(x) = 1,$$

and the above equivalence is true for all possible input lengths $n \in \mathbb{N}$. This is a *non-uniform model of computation*. Recall from the last lecture, that each circuit C_n is comprised of elementary gates ($\neg$, $\vee$, $\wedge$). Large circuits require many elementary gates, while small circuits get by with only a few. This cost is measured by the circuit *size* $s(C_n)$ that counts the total number of elementary gates.

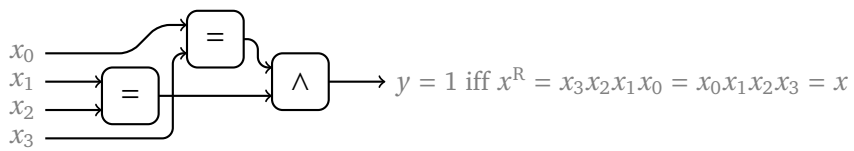
It is reasonable to expect that the circuit size $s(C_n)$ should grow with the number of input bits n. This is reminiscent of the observation that TMs take longer to process large inputs in the sense that the runtime $T(n)$ must grow with input length $n = |x|$. Capturing the rate of growth gave rise to TM complexity classes like **P** (the recognizing TM runtime scales polynomially in input length) and **EXP** (the recognizing TM runtime scales exponentially in input length). For circuit complexity, the crucial question is: given a language A, how fast does the recognizing circuit size grow if we increase input length n? This is captured by the following definition.

size-bounded (circuit) computations

Definition 13.1 (size-bounded (circuit) computations). Let $f : \mathbb{N} \to \mathbb{R}_+$ be a function. We say that a language $A \subseteq \{0,1\}^*$ is in **SIZE**$(f(n))$ if there exists a circuit family $\{C_n\}_{n\in\mathbb{N}}$ such that $s(C_n) \leq O(f(n))$ and for every $x \in \{0,1\}^n$, $x \in A$ if and only if $C_n(x) = 1$.

This is the circuit counterpart of our definition of TM time complexity classes from Lecture 6.

Example 13.2 (PALINDROME $\in$ SIZE(n)). Let us consider the language (binary) PALINDROME $= \{x \in \{0,1\}^* : x^R = x\}$, where x^R is the reverse of bitstring x. The following circuit recognizes palindrome structure among bitstrings of (even) length $n = 4$:



This circuit is comprised of $2 = n/2$ logical equalities ('=') and $1 = n/2 - 1$ $\wedge$-gates. This construction readily generalizes to all inputs of even length n and a slight modification allows for handling odd input lengths as well (ignore the central input bit). If we further decompose each logical equality into 5 elementary gates ($\varphi(a,b) = (a \vee \bar{b}) \wedge (\bar{a} \vee b)$), we obtain a palindrome recognizing circuit family $\{C_n\}_{n\in\mathbb{N}}$ that obeys

$$s(C_n) = 5\lfloor n/2 \rfloor + \lfloor n/2 \rfloor - 1 = 6\lfloor n/2 \rfloor - 1 = O(n).$$

Hence, PALINDROME $\in$ **SIZE**(n). ∎

Next, recall that there is a one-to-one correspondence between languages A and Boolean formulas:

$$f_A(x) = \begin{cases} 1 & \text{if } x \in A, \\ 0 & \text{else if } x \notin A. \end{cases}$$

If we also fix the input length to $n = |x|$, this becomes a logical formula with n input bits and one output bit ($m = 1$). And in the last lecture, we have seen that every logical function $f_n : \{0,1\}^n \to \{0,1\}^m$ can be represented by a circuit C_n. In fact, Theorem 12.5 asserts that this circuit has size (at most) $s(C_n) \leq n2^{n+1}$. Combined with Definition 13.1, this allows us to draw the following conclusion:

Corollary 13.3 Every language $A \subseteq \{0,1\}^*$ is contained in **SIZE** $(n2^n)$.

exponential-size circuits can compute 'everything'

This statement hints at the existence of a circuit family for *every* language conceivable – a curiously strong statement, given that we know that some languages cannot be computed with Turing machines. But, more on that later. For now, we emphasize that Corollary 13.3 only provides size-bounds that scale exponentially in input length n (think **EXP**, but for circuits). The following definition isolates languages that come with circuits that have a more favorable resource scaling.

Definition 13.4 (PSIZE). The problem class **PSIZE** contains all languages that can be decided using *polynomial-size* circuit families. More formally,

PSIZE = circuit analoge of **P**

$$\mathbf{PSIZE} = \bigcup\nolimits_{k\geq 1} \mathbf{SIZE}\left(n^k\right),$$

where **SIZE** $\left(n^k\right)$ has been introduced in Definition 13.1.

PSIZE is the circuit-based analogue of **P** which encompasses all problems that can be solved in polynomial runtime on a Turing machine (TM). In fact, these two problem classes are related. Recall from last lecture, that *any* TM computation on length-n inputs can be represented by a circuit whose size scales (at most) quadratically in the TM runtime $T(n)$. This was the content of Theorem 12.5 and has the following implication.

Corollary 13.5 (P $\subseteq$ PSIZE). Every language $A \subseteq \{0,1\}^*$ that a TM can decide in polynomial runtime (in input length) can also be decided by a polynomial-sized circuit family. In formulas: $A \in \mathbf{P}$ implies $A \in \mathbf{PSIZE}$.

P $\subseteq$ **PSIZE**

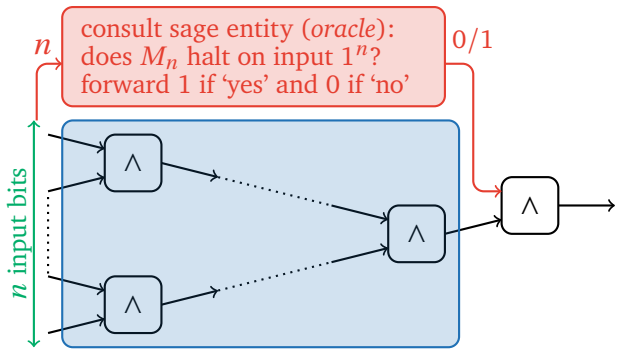


Figure 13.1 *A linear-size circuit family* $\{C_n\}_{n\in\mathbb{N}}$ *that answers the unary halting problem:* $C_n(x) = 1$ if $x = 1^n$ (proper unary encoding) and the TM M_n halts on input 1^n (halting problem); otherwise $C_n(x) = 0$. The circuit contains n $\wedge$-gates ($s(C_n) = n$) and consists of two parts. The blue part merely checks whether the unary encoding is correct. The red part contributes one additional bit that answers the halting problem by hard-wiring the correct answer into the circuit. This is contingent on external advice by a sage and/or powerful entity, often called an *oracle*. This may seem like cheating, but does not violate our definitions of circuits and circuit size.

We see that every language that comes with a polynomial-runtime TM can also be decided by a polynomial-size circuit family. But what about the converse direction: is it possible to verify every problem in **PSIZE** with a polynomial runtime TM? The answer turns out to be *no*: the two complexity classes are actually outrageously different. The discrepancy is best illustrated by the following example which is based on *unary* encodings of natural numbers:

unary encoding of natural numbers

$$n \in \mathbb{N} \text{ is represented as } \quad 1^n = \underbrace{1\cdots 1}_{n \text{ times}} \in \{0,1\}^n .$$

Example 13.6 (the unary halting problem is in PSIZE). Consider a unary encoding of the halting problem from Lecture 5:

the halting problem is in **PSIZE**

$$\widetilde{\text{HALT}} = \{1^n : \text{ the TM } M_n \text{ halts on input } 1^n\} \subseteq \{0,1\}^* . \tag{13.1}$$

In this variant of the language, we encode Turing numbers n in unary and ask whether the associated TM M_n halts for one particular length-n input, namely $x = 1^n$. This variant of the halting problem is known to be *semi-decidable*, i.e. we can use TMs to identify yes-instances, but have no chance to identify no-instances in general. In particular, $\widetilde{\text{HALT}} \notin \mathbf{P}$, because the language class **P** can only contain languages that are decidable.

However, $\widetilde{\text{HALT}} \in \mathbf{PSIZE}$. This is because unary languages only contain one relevant input per input size n. And we can use a single 'hardwired' bit to indicate whether 1^n is part of the language or not. This construction is illustrated in Figure 13.1 and actually covers *all* unary languages. The semi-decidable language introduced in Eq. (13.1) is one particular example. ∎

Example 13.6 identifies a language that is in **PSIZE**, but definitely not in **P** (it's not even computable). This has the following immediate consequence.

Lemma 13.7 (PSIZE $\not\subseteq$ P). There are languages A that are contained in **PSIZE**, but not in **P**. In formulas: **PSIZE** $\not\subseteq$ **P**. **PSIZE** $\not\subseteq$ **P**

We deduced this statement from a rogue example that doesn't quite seem to play according to the rules. As visualized in Figure 13.1, the circuit that decides the halting problem relies on external advice. It does not talk about how this advice is obtained in the first place. And, because of that, the actual description of the circuit seems incomplete and hard (perhaps even impossible) to actually implement. It seems reasonable to restrict our attention to circuit families that have a complete and efficient description in terms of a directed, acyclic graph, see Definition 12.2 from Lecture 12. A single blueprint will not do the job, because different input lengths n necessitate the construction of different circuits C_n. It is more appropriate to demand the existence of an efficient computer program that takes n as an input and provides a blueprint for the associated circuit C_n. This intuition is captured by the following definition.

Definition 13.8 (P-uniform circuit families & languages). A circuit family C_n is *P-uniform* if there exists a polynomial runtime TM which takes 1^n as input and outputs a description of the circuit C_n (e.g. in terms of a bit encoding of the underlying directed, acyclic graph). Likewise, we say that a language $A \subseteq \{0,1\}^*$ is **P**-uniform, if we can decide it with a **P**-uniform circuit family. **P**-uniform circuit families & languages

Note that **P**-uniform circuits always have polynomial size (otherwise, it would take the TM too long to write down an explicit description). The circuit family illustrated in Figure 13.1 is definitely not **P**-uniform. But all other polynomial-size circuits we have encountered so far, are[1]. As soon as we are able to actually write down a circuit family that only contains polynomially many gates (in input length n), we can also automatize this process with a polynomial-runtime TM (this is precisely the meaning of **P**-uniform). Subsequently, we can construct an efficient circuit simulator. That is, a TM that takes a **P**-uniform circuit description as input and simulates its computations on a work tape with only polynomial overhead. When combined sequentially, these two TMs ensure that **P**-uniform circuit computations are contained in **P**. Conversely, the constructive proof of Theorem 12.5 provides a **P**-uniform circuit for every polynomial-runtime TM computation (restricted to a fixed input length). Together, these two implications yield the following equivalence relation.

[1] This claim is contingent on a unary encoding of input length: $|1^n| = n$ ensures that we can afford TM runtimes that scale polynomially in n. If we encoded n in binary instead, the resulting input length $|\llcorner n \lrcorner| = \lfloor \log_2(n) \rfloor + 1$ would be exponentially smaller. And even very simple poly-size circuits, like the size-$O(n)$ circuits from Example 13.2, would take exponential runtime (in $|\llcorner n \lrcorner|$) to write down, because we need $O(2^{|\llcorner n \lrcorner|})$ gates.

Theorem 13.9 (P = P-uniform languages). A language A has **P**-uniform circuits if and only if $A \in \mathbf{P}$.

P = P-uniform languages

To paraphrase: 'sensibly constructed' polynomial-size circuit families and polynomial-time TMs capture the same class of computational problems.

13.2 Turing machines that take advice

We have seen that efficiently constructible circuit families and polynomial-runtime TMs capture the same class of computing problems. And yet, already a single bit of 'magic' information can empower polynomial-size circuits to solve problems that are uncomputable for TMs, see Example 13.6. This interesting observation about circuits has a counterpart for TMs.

P/poly=TMs that take advice

Definition 13.10 (the problem class P/poly). We say that a language $A \subseteq \{0,1\}^*$ is in **P**/poly if membership can be decided by a polynomial-time TM M that also has access to a single, polynomially bounded advice string. More formally, for each input length $n \in \mathbb{N}$ there is a single bitstring a_n with length $|a_n| = \text{poly}(n)$ and a poly(n)-runtime TM M such that $x \in A \Leftrightarrow M(x, a_n) = 1$.

The advice string a_n captures the concept of external advice, e.g. obtained by consulting an *oracle* (a sage entity), as illustrated in Figure 13.1. The following example showcases that this form of advice can be very empowering.

Example 13.11 (all unary languages are in P/poly). A language A is *unary* if it is a subset of $\{1^n : n \in \mathbb{N}\}$. For each input length n, there is only one interesting question: is $1^n = 1\ldots1 \in A$ or not? This binary question can always be answered by a single bit of advice: $a_n = 1$ if $1^n \in A$ and $a_n = 0$ else if $1^n \notin A$. The TM M then simply accepts or rejects based on this single advice bit (which can certainly be done in polynomial time). Remarkably, this class of problems includes uncomputable languages, like $\widetilde{\text{HALT}}$ introduced in Eq. (13.1). ▪

We see that a single advice string can turn uncomputable problems into efficiently computable ones. In particular $\widetilde{\text{HALT}} \in \mathbf{P}/\text{poly}$. Advice for solving a given problem can come in many shapes. For **P**/poly it comes in the form of a bitstring that only depends on input length. For **PSIZE** it comes in the form of a blueprint for a (polynomial-size) circuit that allows us to tackle the problem. The following statement points out that these two forms of advice are actually equivalent (in a theoretical computer science sense).

Theorem 13.12 (P/poly = PSIZE). A language $A \subseteq \{0,1\}^*$ is in **P**/poly if and only if it is also in **PSIZE**. In formulas: **P**/poly = **PSIZE**.

P/poly = PSIZE

We leave the proof as an instructive exercise. Theorem 13.12 relates a natural circuit complexity class (**PSIZE**) to an interesting model of TM computation (**P**/poly).

External advice can be used to model problem-solving strategies that go beyond simply sitting down and developing an algorithm to tackle a certain

challenge. The simplest example is *collaboration* (and also supervision): ask somebody else for advice on how to solve your problem.

machine learning can be modeled as computation with advice

But, there are other, more automated, ways to obtain advice. Several *machine learning* (ML) models also fall into this category. Let us take *binary image classification* – by means of support vector machines or neural networks – as an illustrative example. There, the task is to develop an algorithm that can classify elements of sets into two groups. E.g. does a given image depict a dog or a cat?

Suppose, for simplicity, that all images are comprised of n black/white pixels, i.e. each input can be encoded into a bitstring of length n. Given an input $x \in \{0,1\}^n$, our task is to decide whether the associated picture describes a dog or a cat. If we also assume that every picture we can obtain either corresponds to a dog or a cat, this question boils down to binary decision problem: check if

$$x \in \textsc{Dog}_n = \left\{ x \in \{0,1\}^n : \ x \text{ encodes an image of a dog} \right\}. \tag{13.2}$$

Sitting down and developing an algorithm for this decision problem from first principles is probably very difficult. But this is not what we do in ML. There, we use large amounts of training data – i.e. L labeled pictures (x_l, y_l) of dogs and cats ($y_l = +1$ for dogs and $y_l = -1$ for cats) – to train a labelling function $f_\sharp : \{0,1\}^n \to \{0,1\}$ by minimizing some empirical risk function $R : \mathbb{R} \to \mathbb{R}_+$:

$$\underset{f \in \text{function class}}{\text{minimize}} \quad \sum_{l=1}^{L} R\left(f\left(x_l\right) - y_l\right). \tag{13.3}$$

Different function classes give rise to different ML models. Likewise, different risk functions lead to different optimal functions $f_\sharp$. Choosing the right ML model endowed with the right risk function is an art in itself and would go way beyond the scope of this introductory lecture. Here, we content ourselves by pointing out the following general behavior: the training stage can be very expensive in terms of computational resources. The obtained solution $f_\sharp : \{0,1\}^n \to \{0,1\}$ can, however, typically be stored and computed efficiently (in the number of pixels n). This feature allows us to interpret a binary encoding $\llcorner f_\sharp \lrcorner \in \{0,1\}^{\text{poly}(n)}$ of the obtained ML model as a single advice string a_n that helps us to solve the binary classification problem (13.2) on new inputs. Indeed, it is now easy to devise a TM (algorithm) that executes $f_\sharp$ on a picture $x \in \{0,1\}^n$ we haven't seen yet:

$$M : \ (x, \llcorner f_\sharp \lrcorner) \mapsto f_\sharp(x) \in \{0,1\}.$$

Since $f_\sharp$ is easy to store and compute, this TM operates in polynomial runtime. And if the training stage was successful, $f_\sharp(x)$ should be a good approximation for the true label (cat or dog). While probably not perfect, this TM does stand a good chance at successfully solving the binary classification problem, because it uses additional advice obtained from processing a large number of cat- and dog-images in a separate training stage. The impressive empirical and

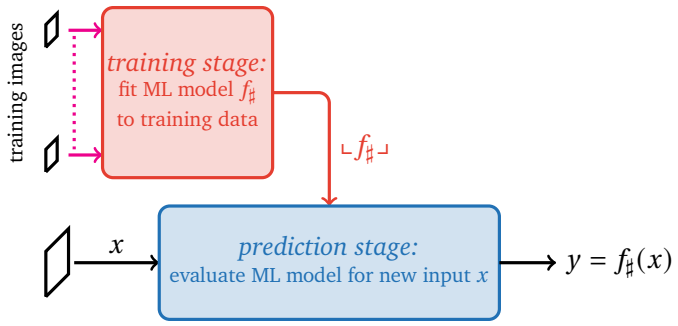


Figure 13.2 *Machine learning as a polynomial-runtime algorithm that takes advice:* On the surface, a machine learning model for binary classification labels n-pixel images $x \in \{0,1\}^n$ by computing an efficient label function $f_\sharp : \{0,1\}^n \to \{0,1\}$ (blue box). This label function, however, is the result of a resource-intensive training stage (red box) that is contingent on access to many labeled training data points. We can model the entire process by a polynomial-runtime TM that has access to a single, polynomial-length advice string $\llcorner f_\sharp \lrcorner$ obtained from the training stage.

theoretical success of ML showcases that this type of advice can make a huge difference. We refer to Figure 13.2 for a visual illustration.

Our analogy between ML and **P**/poly is apt, but not (yet) perfect. When analyzing ML models, one needs to take into account probabilistic models of computation: it is typically enough for a trained ML model to succeed with high probability. And although possible, our deterministic models of computation are too rigid to appropriately capture such probabilistic concepts.

13.3 The Karp-Lipton Theorem

relations between **P**/poly and **NP**

Superficially, our definition of **P**/poly (Definition 13.10) resembles the definition of the problem class **NP** from Lecture 7. Recall that a language A is in **NP** if for every $x \in A$, there exists a polynomial-size y (the 'certificate') and a polynomial-runtime TM M (the 'verifier') such that $M(x,y) = 1$. Conversely, if $x \notin A$, then $M(x,y) = 0$ for all certificates y (soundness). It seems tempting to relate the poly-length advice string a_n in the definition of **P**/poly to poly-length certificates y of membership in the definition of **NP**. But, there are several noteworthy differences:

- **(i)** The certificate of **NP**-membership y depends on the actual input x. In particular, different length-n inputs x can (and in general will) have different certificates. In contrast, problems in **P**/poly must get by with a *single* advice string a_n that must cover all possible length-n inputs simultaneously. This suggests that **NP**-certificates can be more expressive than **P**/poly advice strings.
- **(ii)** Languages A in **NP** are always computable. In particular, for $x \in A$, we can actually compute the associate certificate y in exponential runtime. And,

if we don't find a certificate, soundness ensures that $x \notin A$. The advice string a_n in $\mathbf{P}/\text{poly}$ does not have such a restriction. As Example 13.6 highlights, it can even be uncomputable. Viewed from this perspective $\mathbf{P}/\text{poly}$ seems less restrictive than **NP**.

The two arguments point in opposite directions and a direct comparison between **NP** and $\mathbf{P}/\text{poly}$ seems challenging. Example 13.11, in particular, asserts

$$\mathbf{PSIZE} = \mathbf{P}/\text{poly} \not\subseteq \mathbf{NP},$$

because the former includes variants of the halting problem, while **NP** certainly does not. But what about the converse direction? If the inclusion $\mathbf{NP} \subseteq \mathbf{P}/\text{poly} = \mathbf{PSIZE}$ were true, it would have profound implications on the computing world as we know it. This would imply that every problem that is easy to verify ($A \in \mathbf{NP}$) can also be computed by a circuit family of polynomial size ($A \in \mathbf{PSIZE}$). So, we could in principle build efficient hardware that solves important **NP**-problems, like 3-SAT, traveling salesperson and the like. The 'only' catch is that we may not be able to devise such short circuits efficiently (they are contingent on oracle advice that we don't have access to). But this doesn't have to be a complete dealbreaker. Certain **NP**-problems, take 3-SAT for instance, seem important enough to merit a large, (multi-)government financed research project (think: CERN or the Manhattan project) with the goal of discovering circuit solutions up to, say, $n_{\text{max}} = 100\,000$ input bits by brute force, if need be. Once these circuits are discovered, we could easily integrate them in existing computing hardware. Such special-purpose chips could then be used to efficiently solve **NP**-problems (via reduction to 3-SAT) up to a large threshold input length n_{tot}. While this is not enough to rigorously deduce $\mathbf{P} = \mathbf{NP}$ (we would only know TM simulations of efficient circuits up to threshold size n_{tot}), a practical equality would follow for problems that are not too huge. And, in turn, virtually all practical implications of $\mathbf{P} = \mathbf{NP}$ would carry over as well. E.g. we could jeopardize every encryption scheme that does not use a huge key.

Some of these implications seem utopic and, frankly, too good to be true. So, we might expect that the inclusion $\mathbf{NP} \subseteq \mathbf{PSIZE}$ cannot hold to begin with. The following seminal result by Karp and Lipton provides rigorous evidence in this direction.

Theorem 13.13 (Karp-Lipton Theorem). If $\mathbf{NP} \subseteq \mathbf{PSIZE}$ (equivalently: $\mathbf{NP} \subseteq \mathbf{P}/\text{poly}$), then the polynomial hierarchy collapses to the second level, i.e. $\mathbf{PH} = \mathbf{\Sigma}_2^p$.

$\mathbf{NP} \not\subseteq \mathbf{PSIZE}$ unless **PH** collapses to 2nd level

This theorem states that **NP**-complete problems, like SAT, are even harder than one might think. Circuits, in particular are probably not the answer. Suppose that we were able to develop efficient algorithms that design polynomial-size circuits capable of solving SAT. Then, this would imply $\mathbf{NP} = \mathbf{P}$ (because **P**-uniform circuits are equivalent to **P**). The Karp-Lipton Theorem addresses the next best thing: polynomial-size circuits that solve SAT exist, but may be hard to find. This is captured by the assumption $\text{SAT} \in \mathbf{NP} \subseteq \mathbf{PSIZE} = \mathbf{P}/\text{poly}$.

interpretation: (probably) no small circuits for SAT solving

Karp and Lipton started with this assumption and deduced a collapse of the polynomial hierarchy to the 2nd level. This is the third strongest collapse assertion after **P** = **NP** (0th level collapse) and **NP** = **co-NP** (1st level collapse). So, polynomial-size circuits for SAT solving probably don't exist after all!

The proof of Theorem 13.13 is based on a reduction from $\mathbf{\Pi}_2^p$ to $\mathbf{\Sigma}_2^p$. The first step involves identifying a nice problem that is complete for the class $\mathbf{\Pi}_2^p$.

Proposition 13.14 (A complete problem for $\mathbf{\Pi}_2^p$). The language Π_2SAT contains (binary encodings of) Boolean formulas $\varphi : \{0,1\}^{2n} \to \{0,1\}$ that obey

$$\forall y_0 \in \{0,1\}^n \, \exists y_1 \in \{0,1\}^n \, \varphi(y_0, y_1) = 1. \tag{13.4}$$

This language is complete for the problem class $\mathbf{\Pi}_2^p$ introduced in Lecture 11.

We leave the proof as an exercise and use Proposition 13.14 off the shelf as a starting point for proving the Karp-Lipton Theorem.

Proof of Theorem 13.13. We will use the assumption **NP** $\subseteq$ **PSIZE** to deduce $\mathbf{\Pi}_2^p \subseteq \mathbf{\Sigma}_2^p$. The latter statement is enough to deduce that the polynomial hierarchy collapses to the 2nd level, see Theorem 11.15 from Lecture 11.

To show $\mathbf{\Pi}_2^p \subseteq \mathbf{\Sigma}_2^p$, it is enough to focus on the $\mathbf{\Pi}_2^p$-complete problem Π_2SAT from Proposition 13.14. If we fix $y_0 \in \{0,1\}^n$ in the first quantifier of Eq. (13.4), the remaining part of the formula looks like an instance of SAT:

$$\exists y_1 \in \{0,1\}^n \, \varphi\,(y_0, y_1) = \exists y_1 \in \{0,1\}^n \, \tilde{\varphi}_{y_0}(y_1) = 1.$$

This is where our assumption **NP** $\in$ **PSIZE** comes into play. It guarantees that there is a circuit family $\{C_n\}_{n\in\mathbb{N}}$ with size $s(C_n) = \mathrm{poly}(n)$ such that $C_n\,(\llcorner \varphi \lrcorner, y_0) = 1$ if and only if $\tilde{\varphi}_{y_0}(\cdot) = \varphi\,(y_0, \cdot)$ is satisfiable.

Next, recall from the exercises that we can use yes/no answers to a SAT-problem to actually identify a satisfying assignment of the formula in question (if it exists). The same trick also works for circuits: we can cleverly combine a linear number of C_n-circuits to create a new circuit C'_n that takes input string $(\llcorner \varphi \lrcorner, y_0)$ and produces a n-bit string $v = C'_n(\llcorner \varphi \lrcorner, y_0)$ that is guaranteed to obey $\varphi(y_0, v) = 1$. The size of this new circuit family is still polynomial in n.

Note that our arguments don't provide us with an explicit construction of the C'_n-circuits. We merely know that such circuits must exist and must work for any choice of $y_0 \in \{0,1\}^n$. Moreover, we can encode each poly-size C'_n into a bitstring $\llcorner C'_n \lrcorner$ of polynomial length. In formulas: if **NP** $\in$ **PSIZE**, then truth of Eq. (13.4) must imply

$$\exists \llcorner C'_n \lrcorner \, \forall y_0 \in \{0,1\} \; \varphi\left(y_0, C'_n(\llcorner \varphi \lrcorner, y_0)\right) = 1 \tag{13.5}$$

(and $|\llcorner C'_n \lrcorner| = \mathrm{poly}(n)$). Conversely, if Eq. (13.4) is false, then Eq. (13.5) must be false as well. So, **NP** $\in$ **PSIZE** would imply that Eq. (13.4) and Eq. (13.5) are actually equivalent!

This is beginning to look dangerous. After all, Eq. (13.4) is a complete problem for $\mathbf{\Pi}_2^p$, while Eq. (13.5) looks like a problem in $\mathbf{\Sigma}_2^p$ (an $\exists$-quantifier

followed by a $\forall$-quantifier that hit a Boolean formula of polynomial size). Indeed, it is not difficult to show that the truth of Eq. (13.5) can be evaluated in $\mathbf{\Sigma}_2^p$ (all circuits involved can be represented by bit encodings of polynomial size). This allows us to conclude that **NP** $\subseteq$ **PSIZE** would imply $\mathbf{\Pi}_2^p \subseteq \mathbf{\Sigma}_2^p$. ■

piece of advice: don't use SAT reductions too liberally

We conclude this lecture with a word of caution. The Karp-Lipton Theorem (Theorem 13.13) highlights that SAT is (probably) even harder than one has initially thought. Statements like this remind us to not use SAT-reductions too liberally. In computer science and computer engineering, especially at JKU, it is common practice to take a problem of interest and map it to an instance of SAT. Powerful empirical SAT- and QBF-solvers – several of which have actually been developed by (former) JKU researchers Armin Biere and Martina Seidl – can then be employed to tackle the reduced problem. Statements like **P** $\neq$ **NP** and **NP** $\not\subseteq$ **PSIZE** (which are widely believed to be true) provide strong evidence that such a solution path should not work in full generality. SAT really is a hard problem (at least in the worst case). And the actual problem you started out with might actually be strictly simpler (unless you have already convinced yourself that it is actually **NP**-complete or harder). This might, in particular, be true for problems related to hardware/circuit design, where **PSIZE** = **P**/poly occurs naturally and may cover several interesting problem instances.

Problems

Problem 13.15 (Proof of Theorem 13.9). Prove the following claim: A language A has **P**-uniform circuits if and only if $A \in \mathbf{P}$.

Problem 13.16 (Proof of Theorem 13.12). Prove the following claim: A language $A \subseteq \{0,1\}^*$ is in **P**/poly if and only if it is also in **PSIZE**.

Problem 13.17 (Proof of Proposition 13.14). Prove the following claim: the language Π_2SAT defined in Eq. (13.4) is complete for the problem class $\mathbf{\Pi}_2^p$.
Hint: modify the proof of the Cook-Levin theorem from Lecture 8.

14. Circuit lower bounds & Circuit-SAT

Date: 25 January 2024

Agenda

Today we continue and finalize our circuit-based analysis of computational complexity. In the first part, we will exploit a nice feature of circuits: there is only a finite amount of distinct circuits that we can construct by using a fixed number of s elementary gates. And we can actually count them. Such counting arguments will allow us to rigorously prove reassuring statements, like *(circuit) size does matter*: larger circuits can compute strictly more functions than small ones. Being able to count is another distinct advantage of circuits over Turing machines (TMs). Analogous statements also exist for TMs – longer runtimes allow us to compute strictly more functions – but are harder to establish.

In the second part of this lecture, we revisit one of the central statements of this course from a circuit perspective. The Cook-Levin theorem states that every problem in **NP** can be reduced to an instance of 3-SAT. We already presented a self-contained proof sketch in Lecture 8, but this required honest work. To close the lecture with a bang, we illustrate how a detour via circuit-land yields a much more streamlined proof.

Agenda:

1. Circuit lower bounds
2. Circuit-SAT & and alternative proof of the Cook-Levin Theorem

14.1 Circuit lower bounds

In this section, we explore connections between Boolean formulas $f : \{0,1\}^n \to \{0,1\}$ and Boolean circuits with n inputs and one output ($m = 1$). For each input size n, there are only a finite number of (distinct) Boolean formulas. In fact, we can easily count them.

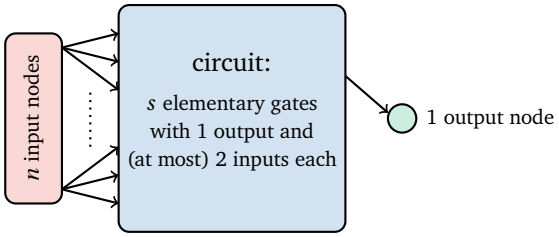


Figure 14.1 *Illustration of a general Boolean circuit geometry:* We can group circuit nodes into three distinct categories. Input nodes (red), a central block (blue) that contains the actual logical gates, as well as a single output node (green).

Proposition 14.1 The number of distinct Boolean functions $f : \{0,1\}^n \to \{0,1\}$ is exactly $N_{\text{fct}}(n) = 2^{2^n}$.

doubly-exponential growth of the number of Boolean functions (in input length n)

Proof. There are a total of 2^n different input configurations. A Boolean formula maps each of them to either 0 or 1. This produces a total of 2^{2^n} choices – two possibilities for each input configuration – each of which produces a distinct Boolean function. ∎

A similar situation arises when studying circuits with one output ($m = 1$). If we fix the number of inputs n, as well as the circuit size s (i.e. the total number of elementary gates), we can also only obtain a finite number of functionally distinct circuits.

Proposition 14.2 The number of distinct size circuits with n inputs, $m = 1$ output and size at most s is upper bounded by $N_{\text{circuit}}(n,s) = n^{2s}(4s)^s$.

exponential growth of the number of circuits (in size s)

This is a rather crude upper bound. As soon as the circuit size s exceeds the number of inputs n ($s \geq n$), we obtain a much simpler functional dependence:

$$N_{\text{circuit}}(n,s) = n^{2s}(4s)^s \leq (4s)^{3s} = s^{O(s)} = 2^{O(s\log(s))}. \tag{14.1}$$

Proof of Proposition 14.2. Consider a general circuit with n inputs, s elementary gates and $m = 1$ output. Then we can always decompose such a circuit into 3 distinct blocks that are visualized in Figure 14.1: one block subsumes all n inputs (left, red), another circuit block contains the s gates (center, blue) and a final block contains the single output node (right, green), see Figure 14.1.

The central part consists of s elementary gates ($\vee$, $\wedge$ or $\neg$) that have exactly 1 output each. These outputs must either lead to another elementary gate or to the output node. This gives a total of $(s-1)+1 = s$ choices per elementary gate. Since there are s elementary gates in total, there can be (at most) s^s different network configurations within the blue and green part of the circuit. Moreover, each of the s gates involved can either be a $\vee$-gate, a $\wedge$-gate or a $\neg$-gate. We also allow for a fourth option – no gate at all – to also cover circuits that have size smaller than s. This freedom of choice produces an additional number of 4^s different gate assignments.

Finally, we must count the number of ways how input bits can be fed into the circuit properly. Each of the s elementary gates has at most 2 inputs. Each of them can correspond to any of the n input bits. So, there are (at most) n^2 different configurations to choose from. Since the circuit is comprised of s elementary gates, this produces a maximum number of n^{2s} different interfaces between input and circuit blocks. Putting everything together produces a maximum circuit count of

$$N_{\text{circuit}}(n,s) \leq n^{2s} \times s^s \times 4^s = n^{2s}\,(4s)^s\,.$$

■

We can now compare the two counts. The total number of Boolean functions grows doubly-exponentially with input length n ($N_{\text{fct}}(n) = 2^{2^n}$). In contrast, the maximum number of circuits only grows (quasi-)exponentially with circuit size s: $N_{\text{circuit}}(n,s) = s^{O(s)} = 2^{s\log(s)}$ provided that $s \geq n$. This exponential discrepancy allows us to derive a powerful lower bound on circuit size.

Theorem 14.3 (circuit lower bounds). 'Almost all' Boolean functions $f : \{0,1\}^n \to \{0,1\}$ require circuit implementations C_f of exponential size (in n): $s\left(C_f\right) \geq 2^n/(6n)$.

'almost all' functions require exponentially large circuits

The attribute 'almost all' is colloquial and deserves further explanation. It means that Boolean functions f that have circuits with the advertised size bound only cover a vanishingly small fraction of all possible Boolean functions. More precisely, we will show that

$$\frac{\text{nr. of } f\text{s that have circuits of size } s(C_f) \leq 2^n/(6n)}{\text{total nr. of Boolean functions with } n \text{ inputs}} < \frac{1}{\sqrt{2^{2^n}}}. \tag{14.2}$$

This bound on the fraction has an appealing probabilistic interpretation. Suppose that we choose a Boolean function $f : \{0,1\}^n \to \{0,1\}$ uniformly at random (from all possible choices). Then, the probability that this function admits a 'short' circuit is doubly exponentially small in n:

probabilistic interpretation of 'almost' all'

$$\Pr_{f:\{0,1\}^n\to\{0,1\}}\left[f \text{ has circuit } C_f \text{ of size } s\left(C_f\right) \leq 2^n/(6n)\right] < \frac{1}{\sqrt{2^{2^n}}}.$$

A quick reformulation $1/\sqrt{2^{2^n}} = 1/2^{2^{(n-1)}}$ highlights that this probability bound indeed decays doubly-exponentially in input size n.

Proof of Theorem 14.3. A given size-s circuit C with n inputs describes exactly one Boolean function $f : \{0,1\}^n \to \{0,1\}$. Let us now set $s_{\max} = 2^n/(6n) \gg n$. Then, Eq. (14.1) tells us that there are at most

$$N_{\text{circuit}}\left(n, s_{\max}\right) \leq \left(4s_{\max}\right)^{3s_{\max}} = \left(\tfrac{4}{6n}2^n\right)^{2^n/(2n)} < \left(2^n\right)^{2^n/(2n)} = 2^{2^n/2}$$

circuits with size $s_{\max}$ or smaller. This is also an upper bound on the number of Boolean functions that can be represented by such circuits (note that, two

or more seemingly distinct circuits can describe the same Boolean function). But $2^{2^n/2} = \sqrt{2^{2^n}}$ is extremely small when compared to the total number $N_{\text{fct}}(n) = 2^{2^n}$ of distinct Boolean functions from Proposition 14.1. In fact, taking the ratio of these two numbers produces the upper bound in Eq. (14.2): $N_{\text{circuit}}(n, s_{\max})/N_{\text{fct}}(n) < \sqrt{2^{2^n}}/2^{2^n} = 1/\sqrt{2^{2^n}}$. ■

Theorem 14.3 provides a lower bound on the circuit size required to represent almost all Boolean functions in n variables. It plays nicely with the deterministic upper bound on maximum circuit size for Boolean functions that we derived in Lecture 12. Applying Theorem 12.5 (universality of circuits) implies that every Boolean function $f : \{0,1\}^n \to \{0,1\}$ (n inputs and $m = 1$ outputs) can be represented by a circuit C_f of size $s\left(C_f\right) \leq n2^{n+1} - 1 \leq (2n)2^n$. This is tantalizingly close to the lower bound we just computed. Hence, almost all Boolean functions have a circuit representation that obeys

two-sided bounds on typical circuit size

$$2^n/(6n) \leq s\left(C_f\right) \leq 2^n \times (2n). \tag{14.3}$$

In this display, lower and upper bound are remarkably close. The corrections $1/(6n)$ on the l.h.s. and $(2n)$ on the r.h.s. are *exponentially* smaller than the leading contribution of 2^n.

We can use Eq. (14.3) to prove that larger circuits can compute strictly more Boolean functions than smaller ones. Such statements are known as *hierarchy theorems* and capture the intuition that increasing computing power – in this case: circuit size – should allow us to compute more things. We content ourselves with establishing a simple statement in this direction. Let $S : \mathbb{N} \to \mathbb{N}$ be a function. Recall that the problem class $\mathbf{SIZE}\left(S(n)\right)$ contains all languages A that can be decided by a circuit family whose size grows as $O\left(S(n)\right)$ (Definition 13.1 from Lecture 13).

(circuit) size does matter for computing power

Theorem 14.4 (hierarchy theorem for circuit sizes (special case)). There are n-variable Boolean functions that can be computed by circuits of size $O(n^2)$, but cannot be computed by circuits of size $O(n)$. In formulas: $\mathbf{SIZE}\left(n\right) \subset \mathbf{SIZE}(n^2)$ (strict inclusion).

Proof. Consider Boolean functions $g : \{0,1\}^n \to \{0,1\}$ that only act nontrivially on the first ℓ input bits. The remaining inputs are ignored. Such functions only have ℓ nontrivial input bits and we can adjust Eq. (14.3) accordingly (replace n by ℓ). If we set $\ell = 1.5\log(n)$, the two bounds tell an interesting story. The upper bound is valid for all functions g and asserts

$$g \in \mathbf{SIZE}\left(2^\ell \times (2\ell)\right) = \mathbf{SIZE}\left(3n^{1.5}\log(n)\right) \subseteq \mathbf{SIZE}\left(n^2\right),$$

because $3n^{1/5}\log(n) = O(n^2)$. Almost all functions g also obey the lower bound in Eq. (14.3). Choose one of them to conclude

$$g \notin \mathbf{SIZE}\left(2^\ell/(6\ell)\right) = \mathbf{SIZE}\left(n^{1.5}/(9\log(n))\right) \quad \Rightarrow \quad g \notin \mathbf{SIZE}\left(n\right),$$

because $n = O\left(n^{1.5}/(9\log(n))\right)$. Hence, there are (many) functions that can be computed in quadratic size, but not in linear size. ■

The proof argument above can be adjusted to cover all types of different size functions.

Theorem 14.5 (hierarchy theorem for circuit sizes (general case)). Let $S, S' : \mathbb{N} \to \mathbb{N}$ be two functions that obey $S'(n), S(n) \leq 2^n/(100n)$ and fix $\epsilon > 0$. Then,

$$S'(n) \geq S(n)\log^{1+\epsilon}(S(n)) \Rightarrow \mathbf{SIZE}\left(S(n)\right) \subset \mathbf{SIZE}\left(S'(n)\right) \tag{14.4}$$

(strict inclusion).

size hierarchy theorem: larger circuit sizes yield strictly more computing power

We leave a rigorous proof as an instructive, if somewhat tedious, exercise. In words, Eq. (14.4) showcases that *(circuit) size does matter*: larger circuit sizes yield strictly more computing power.

A similar hierarchy theorem can be derived for Turing machine (TM) runtimes. Recall that the problem class $\mathbf{DTIME}(T(n))$ contains all languages that can be decided by a TM with runtime $O\left(T(n)\right)$ (Definition 6.7 from Lecture 6).

Theorem 14.6 (hierarchy theorem for TM runtimes (general case)). Now, let $T, T' : \mathbb{N} \to \mathbb{N}$ be two functions and fix $\epsilon > 0$. Then,

$$T'(n) \geq T(n)\log^{1+\epsilon}(T(n)) \Rightarrow \mathbf{DTIME}\left(T(n)\right) \subset \mathbf{DTIME}\left(T'(n)\right) \tag{14.5}$$

(strict inclusion).

time hierarchy theorem: longer TM runtimes yield strictly more computing power

In words, Eq. (14.5) states that *(TM) runtime does matter*: longer runtimes yield strictly more computing power.

Although remarkably similar, the proofs of these two hierarchy theorems are very different. Eq. (14.4) ultimately follows from a counting argument over all possible circuit configurations of a given size. Such counting arguments don't work for TM runtimes, because TMs can loop. Eq. (14.5) instead follows from a contradiction argument that is reminiscent of our proof that the halting problem is uncomputable (see Lecture 5). We refer to [AB09, Section 3.1] for details.

14.2 CIRCUIT-SAT & an alternative proof of the Cook-Levin theorem

By now, we have seen several examples that showcase how circuit-based arguments provide a new perspective on the study of computational complexity. Intuitive concepts like parallelized runtime (which is captured by circuit depth, see Lecture 12) and computation with advice (which captures small circuits with intricate blueprints, see Lecture 13) arise naturally when studying circuits, but take quite some time and effort to properly capture in the TM model. But

circuits also provide us with new tools to discover powerful statements about computational power. The size hierarchy theorem displayed in Eq. (14.4) comes to mind. In addition, our current understanding of circuits also allows us to re-derive other deep results about computational complexity. Let us illustrate this for one of the key results presented in this lecture – the *Cook-Levin Theorem* which we already discussed in Lecture 8.

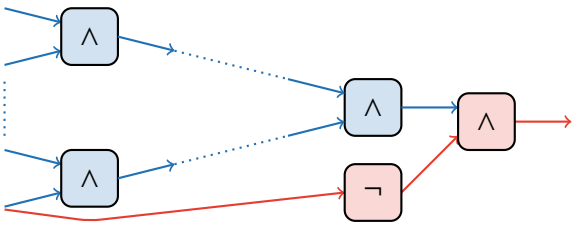
Informally speaking, the Cook-Levin theorem states that every **NP** problem can be reduced to an instance of 3-SAT with only a polynomial overhead. Or, in the jargon from Lecture 9: 3-SAT is **NP**-complete. We refer to these two lectures for a detailed discussion what these two statements formally mean. For now, it is enough to recapitulate that our original proof sketch of the Cook-Levin theorem had been quite involved. It turns out that circuits provide a short cut that allows us to come up with a much more streamlined proof. This new argument uses a detour over a circuit-variant of the Boolean satisfiability problem.

Definition 14.7 (Circuit-SAT). We say that a (Boolean) circuit $C : \{0,1\}^n \to \{0,1\}$ with n inputs and $m = 1$ output is *satisfiable*, if there exists an input string $x \in \{0,1\}^n$ such that $C(x) = 1$. The language Circuit-SAT consists of all (proper) bit encodings of Boolean circuits that are satisfiable:

Circuit-SAT

$$\text{Circuit-SAT} = \{\llcorner C \lrcorner : \ C \text{ is a Boolean circuit that is satisfiable}\} \subseteq \{0,1\}^* .$$

Example 14.8 Consider the following circuit with n inputs and one output ($m = 1$) that is comprised of two parts:



The blue circuit part C_{blue} is is satisfiable, because it computes an AND of all n inputs. Setting $x_0 = \cdots = x_{n-1} = 1$ yields $C_{\text{blue}}(x) = 1$. Including the red part as well yields a circuit $C_{\text{blue+red}}$ that is not satisfiable anymore: $C_{\text{blue+red}}(x) = x_0 \wedge \cdots \wedge x_{n-1} \wedge \bar{x}_{n-1} = 0$ for all possible choices of $x_0, \ldots, x_{n-1}$ ($x_{n-1} \wedge \bar{x}_{n-1} = 0$ for both $x_{n-1} = 0$ and $x_{n-1} = 1$). ∎

Circuit-SAT is actually an important problem in its own right. Many challenges in hardware design can be reduced to an instance of this decision problem. Discussing these relations would go beyond the scope of this introductory lecture. For the task at hand – prove the Cook-Levin theorem – the following rigorous statement suffices.

Theorem 14.9 (Circuit-SAT is NP-hard). Every **NP**-problem admits a polynomial-time reduction to an instance of Circuit-SAT.

Circuit-SAT is **NP**-hard

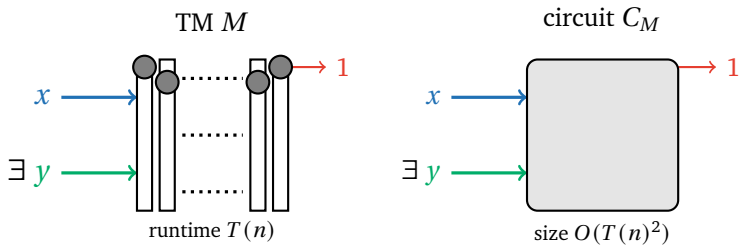


Figure 14.2 *Circuit-SAT is NP-hard:* All **NP** languages A have a TM verification procedure *(left)*: $x \in A$ if and only if $\exists y$ s.t. $M(x, y) = 1$ and the TM runtime obeys $T(n) = \text{poly}(n)$, where $n = |x|$. We can now use Theorem 12.10 from Lecture 12 to represent the entire TM computation as a circuit C_M of size $O(T(n)^2) = \text{poly}(n)$ *(right)*. If we fix the input x, this poly-size circuit is satisfiable if and only if $x \in A$. In formulas: $C_M(x, \cdot) \in$ Circuit-SAT $\Leftrightarrow x \in A$.

This is a powerful statement that resembles the Cook-Levin theorem itself – albeit with Circuit-SAT instead of 3-SAT. Regarding the proof, we have actually already done most of the heavy lifting in Lecture 12. Theorem 12.10 states that every polynomial-runtime TM M can be represented by a Boolean circuit C_M of polynomial-size. Theorem 14.9 is almost an immediate consequence of this reformulation, see Figure 14.2 for a visual illustration.

Proof of Theorem 14.9. Suppose that $A \subseteq \{0,1\}^*$ is a language in **NP**. Then, by definition, we must be able to efficiently certify membership. More formally, there exists a polynomial-runtime TM M (the 'verifier') such that $x \in A$ if and only if $M(x, y) = 1$ for some bitstring $y \in \{0,1\}^{n'}$ (the 'certificate'). Moreover, $n' = \text{poly}(n)$, where $n = |x|$ (the certificate must be short). In formulas:

$$x \in A \Leftrightarrow \exists y \in \{0,1\}^{n'} \text{ such that } M(x, y) = 1. \tag{14.6}$$

We can now use Theorem 12.10 to convert the TM M (for fixed input lengths) into a Boolean circuit C_M with two input strings $C_M : (x, y) \mapsto M(x, y)$. In addition, we can hardwire fixed x-inputs into the circuit geometry to obtain a modified circuit $C_{M,x} : y \mapsto M(x, y)$ that now only depends on the y-input. Doing so allows us to replace Eq. (14.6) by

$$x \in A \Leftrightarrow \exists y \in \{0,1\}^{n'} \text{ such that } C_{M,x}(y) = 1. \tag{14.7}$$

The right hand side of this modified display is an instance of Circuit-SAT. Given M and x, we can actually construct a description of the polynomial-size circuit $C_{M,x}$ in polynomial time. This follows from the fact that we proved Theorem 12.10 by presenting an explicit and efficient construction. ■

The second step of our argument involves establishing a hardness relation between Circuit-SAT and 3-SAT.

Lemma 14.10 (Circuit-SAT $\leq_p$ 3-SAT). Every Circuit-SAT instance admits a linear-time reduction to an instance of 3-SAT. Circuit-SAT reduces to 3-SAT

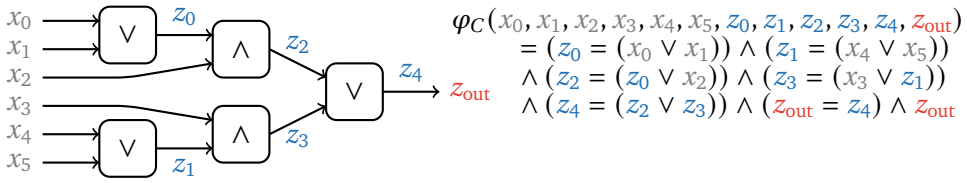
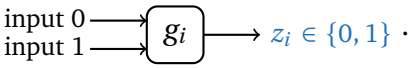


Figure 14.3 *Example for reducing* Circuit-SAT *to 3-SAT:* We convert a Boolean circuit with size $s = 5$ and $n = 6$ inputs *(left)* into a Boolean formula with $n+s+1 = 12$ variables and $s+2$ clauses *(right)*. This is achieved by introducing Boolean variables for inputs (gray), outputs of elementary gates (blue) and the output bit (red). The Boolean formula φ checks that all elementary gates have been implemented correctly (logical equalities between gate inputs and gate output) and that the output bit is 1 (true). Further Decomposing logical equalities as $(a = b) = (a \vee \bar{b}) \wedge (\bar{a} \vee b)$ produces a 3-CNF with (at most) $3(s+1)$ 3-clauses. This 3-CNF is satisfiable if and only if the original circuit is.

Proof. The reduction is visualized in Figure 14.3. Here, we supply additional details, as well as a general construction. Let $C : \{0,1\}^n \to \{0,1\}$ be a Boolean circuit with size s. It suffices to show that the underlying functionality can be represented by a 3-CNF φ with $n' = \text{poly}(n,s)$ variables and $l' = \text{poly}(n,s)$ clauses. Conveniently, the circuit C itself almost gets us there, because it describes a composition of n elementary gates that have (at most) 2 inputs and 1 output each ('circuit computation is local'). We introduce a new dummy variable for each of the s elementary gates:



We can now use logical equalities ('=') to enforce the correct execution of each such gate:

$$\begin{aligned} g_i = \wedge: &\quad z_i = (\text{input } 0 \wedge \text{input } 1), \\ g_i = \vee: &\quad z_i = (\text{input } 0 \vee \text{input } 1) \quad \text{and} \\ g_i = \neg: &\quad z_i = (\neg \text{input } 0), \end{aligned}$$

because $\neg$ only has a single input we need to consider. Subsequently, we can rewrite logical equality as $(a = b) = (a \vee \bar{b}) \wedge (\bar{a} \vee b)$ to further decompose each of the above relations into a 3-CNF comprised of (at most) 3 clauses. Our final formula φ_C has a total of $n' = (n+s+1) = O(n) + O(s)$ input variables:

$$(x_0, \ldots, x_{n-1}, z_0, \ldots, z_{s-1}, z_{\text{output}}).$$

The first n variables denote input bits, the central block contains s consistency-check bits and the final bit records the value of the output node. The formula φ_C itself and corresponds to taking the AND of all consistency enforcements AND the final output variable. This produces a 3-CNF with (at most) $l' = 3(s+1) = \mathcal{O}(s)$

clauses, i.e. the clause number of φ_C is linear in the size s of the original circuit. The first $3s$ clauses ensure that the circuit computation is executed correctly and isolating the last variable z_{output} using 2 size-1 clauses ensures that the formula φ_C is satisifable if and only if the circuit C_M is. ■

re-derivation of the Cook-Levin Theorem

The Cook-Levin theorem is now an immediate consequence of Theorem 14.9 (**NP** $\leq_p$ Circuit-SAT) and Lemma 14.10 (Circuit-SAT $\leq_p$ 3-SAT).

Corollary 14.11 (Cook-Levin Theorem). 3-SAT is **NP**-complete, i.e. every **NP**-problem admits a polynomial-time reduction to an instance of 3-SAT.

Bibliography

[AKS04] M. Agrawal, N. Kayal, and N. Saxena. "PRIMES is in P". In: *Ann. of Math. (2)* 160.2 (2004), pages 781–793. ISSN: 0003-486X. DOI: 10.4007/annals.2004.160.781. URL: https://doi.org/10.4007/annals.2004.160.781.

[AB09] S. Arora and B. Barak. *Computational complexity*. A modern approach. Cambridge University Press, Cambridge, 2009, pages xxiv+579. ISBN: 978-0-521-42426-4. DOI: 10.1017/CBO9780511804090. URL: https://doi.org/10.1017/CBO9780511804090.

[Kar72] R. M. Karp. "Reducibility Among Combinatorial Problems". In: *Complexity of Computer Computations*. The IBM Research Symposia Series. Plenum Press, New York, 1972, pages 85–103. URL: http://cgi.di.uoa.gr/~sgk/teaching/grad/handouts/karp.pdf.

[KS17] C. Kingsford and D. Sleator. *15-451/561: Algorithms, Lec. 23: NP-completeness (lecture notes)*. 2017. URL: https://www.cs.cmu.edu/~15451-f17/lectures/lec23-np.pdf.

[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.

[Way01] K. Wayne. *COS 423 Analysis of Algorithms Lectures: Polyonmial time reductions (lecture notes)*. 2001. URL: https://www.cs.princeton.edu/~wayne/cs423/lectures/reductions-poly-4up.pdf.