WRITE-ONLY DRAFT, DON'T DISTRIBUTE

Computability and Introduction to Complexity: DRAFT Lecture Notes

Edward A. Hirsch*

Abstract

These are lecture notes for the course "Computability and Introduction to Complexity" given by Edward A. Hirsch and Anat Paskin-Cherniavsky in Winter Semester 2025/26 at Ariel University. It assumes the knowledge of the latest version of the Automata (Computational Models) course as well as of the Algorithms-2 course. The most crucial definitions from these courses will be reminded.

Caution: these notes are writen in a write-only style. This is not a book, certainly there are typos. If you spot a problem here, please report to Edward.

Contents

1 Notation and reminders from the previous courses 3

1.1 Languages, problems, encodings, DTMs . . . 3

1.2 Complexity and computability classes . . . 5

2 The deterministic time hierarchy 10

2.1 A more formal treatment of the **DTime** hierarchy . . . 12

3 Reductions and completeness 14

3.1 Many-one reductions (without a time bound), **RE**-completeness, negative results 14

3.2 Polynomial-time many-one reductions and **NP**-completeness . . . 18

3.3 Other types of reductions . . . 21

3.4 Padding . . . 22

4 Rice's theorem and more undecidabile languages 23

*Ariel University

5 The Arithmetical Hierarchy **25**
5.1 Classes defined using oracles . . . 25
5.2 The oracle-style definition of **AH** . . . 26
5.3 An equivalent definition of **AH** using quantifiers . . . 26
5.4 Examples of languages in **AH** . . . 29
5.5 Complete problems . . . 29

6 RE-intermediate languages **31**
6.1 Immune and simple sets . . . 31
6.2 Productive sets . . . 33

7 Gödel's incompleteness theorem — a computability point of view **35**

8 Introduction to Kolmogorov complexity **37**

9 Search to decision reductions **40**
9.1 Search problems . . . 40
9.2 A search-to-decision reduction for `Circuit-SAT` . . . 41
9.3 Search-to-decision reductions for other languages . . . 42

10 The Cook-Levin Theorem: A proof **43**

11 The Polynomial Hierarchy **45**
11.1 Two equivalent definitions . . . 46
11.2 Complete problems for the classes of the Polynomial Hierarchy . . . 50
11.3 **PH** altogether . . . 51
11.4 Collapsing the Polynomial Hierarchy . . . 51

12 Polynomial space: The class PSPACE **53**
12.1 A motivating example . . . 53
12.2 Space complexity classes — a formal definition . . . 54
12.3 Quantified Boolean formulas — a **PSPACE**-complete problem . . . 55
12.4 **PSPACE** on our map . . . 58

13 Randomized computation **60**
13.1 One-sided bounded error . . . 60
13.2 Two-sided bounded error . . . 64
13.3 Another view of **RP** and **BPP** . . . 65
13.4 Errorless randomized algorithms . . . 66
13.5 Observations: Where randomized algorithms are on the map? . . . 68

1 Notation and reminders from the previous courses

1.1 Languages, problems, encodings, DTMs

A language is a set of strings (or “words”), where a string is a finite sequence of symbols (or “letters”) belonging to a finite alphabet (which is a finite set of “letters”). The most important alphabet is the binary alphabet $\{0, 1\}$. **The binary alphabet is the default alphabet for this course.**

The decision problem associated with a language L is a computational problem where you (or a computational device or an algorithm) are given an input x and you are asked to answer whether $x \in L$ (“yes” or “no”).

We consider interesting mathematical problems (like graph coloring or the acceptance problem for a Turing machine), and they need to be encoded in our binary alphabet. We consider reasonable encodings, which means that

- The encoding is not oversized (no exponential-size encodings when smaller encodings are possible — if we are talking about complexity).
- The syntax is easy to check. In particular, it means that every input is mapped to some object: syntactically incorrect inputs are mapped to a specific fixed object (for example, for the Boolean satisfiability problem `SAT` one can map syntactically incorrect encodings to the trivial formula `False` without variables; thus the languages of satisfiable formulas `SAT` and unsatisfiable formulas `UNSAT` are the complements of each other).
- Natural features of our objects are easily (polynomial-time) extractable from the encoding. For example, for an undirected graph it is easy to list all the neighbours of a vertex.
- Unnatural features of our object are not included in the encoding (for example, a Boolean formula does not come with its satisfying assignment, if any).

In particular, every DTM has a binary encoding; if we prepend it with 1 in the beginning, we get a natural number (in binary), called a Turing number of the machine. Why “a” and not “the”? Because for DTMs we consider encodings such that every machine has infinitely many representations (think of trailing zeroes at the end of the encoding that do not change the machine itself, only the length of its description).

Another important encoding is the encoding of pairs. Think how many extra bits you need to encode a pair of bit strings. One can easily encode a pair (a, b) using additional $2(|a| + |b|)$ bits, but actually $O(\log_2 |a|)$ extra bits suffice: write $1^{\lceil \log_2 |a| \rceil}$, then write 0, then write $\lceil \log_2 |a| \rceil + 1$ bits of the binary representation of $|a|$, then write a, then write b — after reading the first parts of this encoding, it is clear where the next part starts, so we can extract a and extract b.

Specific reasonable encodings usually do not matter. When they do matter, we say this explicitly. We can use angular brackets $\langle \ldots \rangle$ to emphasize we are talking about the encoding of the object in the brackets, but otherwise when there is no doubt we will omit them.

The main computation model in this course is the deterministic Turing machine (DTM), which keeps its memory on a tape (an infinite array of symbols) or several tapes. For the most part of the course one can think of DTMs simply as algorithms (namely, deterministic algorithms — no randomness, parallel threads or anything like that). We will remind fine details of the definition when we need them, but the most important points are:

- A DTM can work on inputs of any size (like a normal algorithm): there is a single device (algorithm) for all the queries, we do not need separate devices for smaller and for larger inputs.

- A DTM M can accept (denoted $M(x) = 1$, the answer yes), reject (denoted $M(x) = 0$, the answer no), or otherwise it does not stop (denoted $M(x) = \nearrow$, some books use the notation $M(x) = \infty$). DTMs can also compute functions (not just 0/1 answers), we will come to this when we need it.

- DTMs can have any (constant) number of tapes and work over any alphabet, but in the previous course we proved that one can assume the input alphabet $\{0, 1\}$, the tape alphabet $\{0, 1, _, \triangleright\}$

- In the previous course we built a universal Turing machine (UTM) U that can simulate any DTM (with any number of tapes), that is, for every input $x \in \{0, 1\}^*$, $U(M, x) = M(x)$ (in particular, if $M(x)$ does not stop, $U(M, x)$ does not stop either).

- Moreover, U is efficient enough. One can build such a machine with the running time $O(t(n) \log t(n))$, where $t(n)$ is the running time of $M(x)$ for $|x| = n$, but what we proved in the previous course is that the running time of $U(M, x)$ is polynomial w.r.t. the running time of $M(x)$. This is a fixed polynomial (the degree is a bit different depending on whether you want your UTM to have a single tape or two tapes), and it is enough for us in this course.

DTMs can solve decision problems (languages) in two ways.

A machine M decides a language L if it always stops and $\forall x \in \{0, 1\}^* \; (\, M(x) = 1 \iff x \in L\,)$.

A machine M recognizes (accepts) a language L if it stops and accepts on every word in this language; for other words it can either reject or work indefinitely long (i.e., it does not stop). Some books call such machines "semi-algorithms" to emphasize they solve only one half of the problem.

We denote by $L(M)$ the language that is accepted by a DTM, that is, $L(M) = \{x \in \{0, 1\}^* : M(x) = 1\}$. Note that if a machine decides a language, it also accepts it, but not necessarily vice versa.

1.2 Complexity and computability classes

A class of languages is a subset of $2^{\{0,1\}^*}$ (the set of all sets of bit strings of finite length).

Computability classes. The class of all recursive languages (denoted $\mathbf{R}$) includes all the languages that can be decided by a DTM.

The class of all recursively enumerable languages (denoted $\mathbf{RE}$) includes all the languages that can be recognized (accepted) by a DTM.

Deterministic complexity classes. The running time of a machine M on a certain input x is denoted $\mathrm{time}_M(x)$, this is the number of steps that M performs on input x before reaching the final state. The worse-case running time

$$\text{wc-time}(n) = \max_{x\in\{0,1\}^n} \mathrm{time}_M(x)$$

is a function $\mathbb{N} \to \mathbb{N}$. We will be typically interested in the asymptotic behaviour of the latter function (it is usually characterized up to $O(\dots)$ or even up to a polynomial).

Let us define the classes **DTime** of decision problems that can be solved within a certain time bound: Let $f : \mathbb{N} \to \mathbb{N}$ be a function. We say that a language L belongs to the class $\mathbf{DTime}[f(n)]$ if there exists a (multi-tape) DTM M that decides L in time $O(f(n))$.

Some books use here $O(f(n))$ by default, some prefer strict $f(n)$. We will emphasize it when it is important.

The class of polynomial-time decidable languages is called $\mathbf{P}$. We define it formally as

$$\mathbf{P} = \bigcup_{k\in\mathbb{N}} \mathbf{DTime}[O(n^k)].$$

One can define a larger class

$$\mathbf{EXP} = \bigcup_{k\in\mathbb{N}} \mathbf{DTime}[O(2^{n^k})]$$

and even larger classes like

$$\mathbf{EEXP} = \bigcup_{k\in\mathbb{N}} \mathbf{DTime}[O(2^{2^{n^k}})].$$

All of them are trivially included in $\mathbf{R}$.

Nondeterministic classes and the existential quantifier. A nondeterministic Turing machine N makes a nondeterministic choice at each step. One can assume that this choice is always between two alternatives 0 or 1, thus one can say that a machine guesses bits. We are only interested in the case ("computational path") when the guesses were successful, that is, the machine eventually accepted. If such a sequence of guesses exists for an input x,

we say that it accepts, $N(x) = 1$. If there is no such sequence, but the machine stops and rejects for every sequence of guesses, we say that it rejects, $N(x) = 0$. Otherwise the answer is indefinite, $N(x) = \nearrow$ (that is, for some sequences of guesses it does not stop, and there is no sequence where it accepts).

The running time of a NTM is the maximum running time over all possible nondeterministic choices.

In the polynomial-time case a NTM always stops, thus there is no need to argue about infinite paths. Otherwise we must take care of them.

We remind two definitions for the **NP** class (in the previous course we proved the equivalence between them).

Definition 1.1 (**NTime**, **NP**)**.** We say that L belongs to the class $\mathbf{NTime}[f(n)]$ if there is an NTM N such that

- $\forall x \in \{0,1\}^* N(x) = 1 \iff x \in L$,
- $\forall n \in \mathbb{N}$ wc-time$_N(n) = O(f(n))$.

(Again, some books reserve the notation $\mathbf{NTime}[f(n)]$ for the exact number of steps $f(n)$ and not $O(f(n))$.) We define $\mathbf{NP} = \bigcup_{k \in \mathbb{N}} \mathbf{NTime}[O(n^k)]$.

Recall that a binary relation S on a set T is a set of pairs, or a subset of $T \times T$. We say that $S(x, w)$ is true if $(x, w) \in S$; it is false if $(x, w) \notin S$.

For a pair (x, w), we call x a question, or an instance of the problem S, and we call w a candidate solution, or a witness.

One can define **NP** based on a polynomial-time verifiable binary relation $S \subseteq \{0,1\}^* \times \{0,1\}^*$, that is, on a relation $S \in \mathbf{P}$. (Note that a relation is a set of pairs, thus — if encoded reasonably — a language consisting [the encodings of] all the pairs $\langle x, w \rangle$ such that $S(x, w)$ is true.)

Definition 1.2 (**NP**, through relations)**.** A language L belongs to the class **NP** if there is $S \in \mathbf{P}$ and a polynomial p such that for every $x \in \{0,1\}^*$,

$$x \in L \iff \exists w \in \{0,1\}^{\leqslant p(n)} \ \langle x, w \rangle \in S$$

(Note that the condition implies also that $x \notin L \iff \forall w \in \{0,1\}^{\leqslant p(n)} \ \langle x, w \rangle \in \overline{S}$.)

We emphasize once again that **the existential quantifier** $\exists$ is what defines a nondeterministic computation.

In the Automata course we had also an "intermediate" definition through a "NTM version 2": a DTM with a distinguished "witness" tape, where the witness is written on it (by someone) in the starting configuration. It is obviously equivalent to the definition through relations; in this course we do not use it.

An important example of an **NP** problem that you definitely heard of is

Example 1.1 (SAT)**.** Satisfiable Boolean formulas in conjunctive normal form (CNF):

$$\texttt{SAT} = \{F \text{ in CNF} : \exists x\ F[x] = \texttt{True}\}.$$

Think about an NTM that decides this language!

In the world of computability classes there is a similar characterization (with the existential quantifier) for **RE** (which we also proved in the Automata course):

Theorem 1.1. *$L \in \mathbf{RE}$ if and only if there exists a language $S \in \mathbf{R}$ such that $\forall x \in \{0,1\}^*$*

$$x \in L \iff \exists w \in \{0,1\}^*\ \langle x, w\rangle \in S$$

Classes of complements.

Definition 1.3 (co-)**.** For a class $\mathcal{C}$, denote

$$\mathbf{co}\text{-}\mathcal{C} = \{L : \overline{L} \in \mathcal{C}\}.$$

As simple as that. Note that $\mathbf{co}\text{-}\mathcal{C}$ is **absolutely not** $\overline{\mathcal{C}}$!

If one uses this definition formally together with Def. 1.2 and Th. 1.1 by taking the negation (because we are know talking about x that is outside an **NP** or an **RE** language, respectively), one arrives at alternative characterizations of **co-NP** and **co-RE**:

$L \in \mathbf{co}\text{-}\mathbf{RE}$ if and only if there exists a language $S \in \mathbf{R}$ such that $\forall x \in \{0,1\}^*$

$$x \in L \iff \forall w \in \{0,1\}^*\ \langle x, w\rangle \in S. \tag{1}$$

$L \in \mathbf{co}\text{-}\mathbf{NP}$ if there is a language $S \in \mathbf{P}$ and there is a polynomial p such that for every $x \in \{0,1\}^*$,

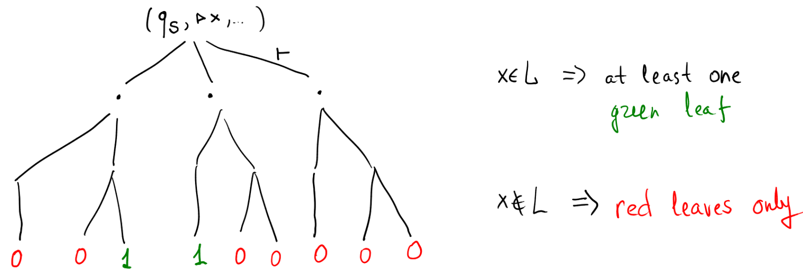
$$x \in L \iff \forall w \in \{0,1\}^{\leqslant p(n)}\ \langle x, w\rangle \in S.$$

Note that the language S here is in fact the complement of the language S in our definitions for **RE** and **NP**, but since the classes **R** and **P** are closed under the complement, it does not matter.

For **co-RE**, we proved that $\mathbf{R} = \mathbf{RE} \cap \mathbf{co}\text{-}\mathbf{RE}$: if one has a machine accepting L and another machine accepting $\overline{L}$, one can write these two machines "in parallel" (interleaving their steps) until one of them accepts — now we know the answer for sure. By the definition of **RE**, one of them has to accept (and thus it stops!).

The situation for **co-NP** is not so clear. We all know that $\mathbf{P} =^? \mathbf{NP}$ is a major open quesion of computational complexity. The question $\mathbf{NP} =^? \mathbf{co}\text{-}\mathbf{NP}$ is the central problem of propositional proof complexity (can one give a polynomial-size polynomial-time verifiable proof that a Boolean formula is unsatisfiable?). We do not know whether $\mathbf{P} = \mathbf{NP} \cap \mathbf{co}\text{-}\mathbf{NP}$ either (this question is somewhat important for cryptography, for example, public-key encryption, — think why).

Computation tree is a very important notion. The computation of a specific NTM N on a specific input x is best viewed as a tree:



Each node of this tree is labeled by a configuration. The root corresponds to the starting configuration on the input x. An edge goes from a configuration C down to a configuration C' on the next level if $C \vdash C'$. Since our machine is nondeterministic, there may be more then one edge going from a node.

A computational path is a path from the root to a leaf. It corresponds to a run of the machine according to specific nondeterministic choices.

An accepting path is a path finishing in q_Y. A rejecting path is a path finishing in q_N.

It is easy to see that if N decides L, then

- $x \in L$ iff $N(x)$ has at least one accepting path,
- $x \notin L$ iff all the paths of $N(x)$ are rejecting.

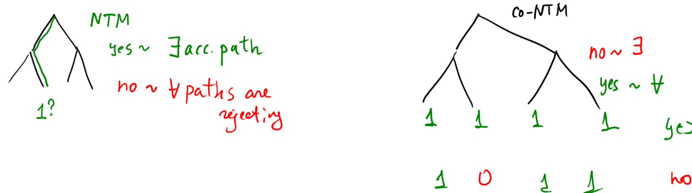
Remark 1.1. The most important thing here is the *quantifier*: in the positive case it is $\exists$ ("at least one"), and in the negative case it is $\forall$ ("all the paths"). Note that if we exchange the words "accepting" and "rejecting" in the two items above, the things will not change much: it will be easy to replace N by another N' that will do the job (just exchange q_Y and q_N). However, there is no easy way to change the quantifier.

Similarly to other "**co-**" classes, one can consider **co-NP**, the class of complements to the languages in **NP**:

$$\textbf{co-NP} = \{L : \overline{L} \in \textbf{NP}\}.$$

These languages can be decided by co-nondeterministic machines that are just nondeterministic machines (NTM), but the semantics for saying "yes" and "no" is different:

Now $x \in L$ if all the computational paths are accepting, and $x \notin L$ if at least one path is rejecting.



Observe again that while it is easy to exchange the accepting and the rejecting states of a machine (the answer "yes" and "no" for a single computational path), it won't give an NTM (with the usual semantics) for the complement of an **NP** language: the quantifiers will be the opposite. Instead, it will give a co-nondeterministic machine for it.

Two important examples of **co-NP** problems are

Example 1.2.

- Unsatisfiable formulas:

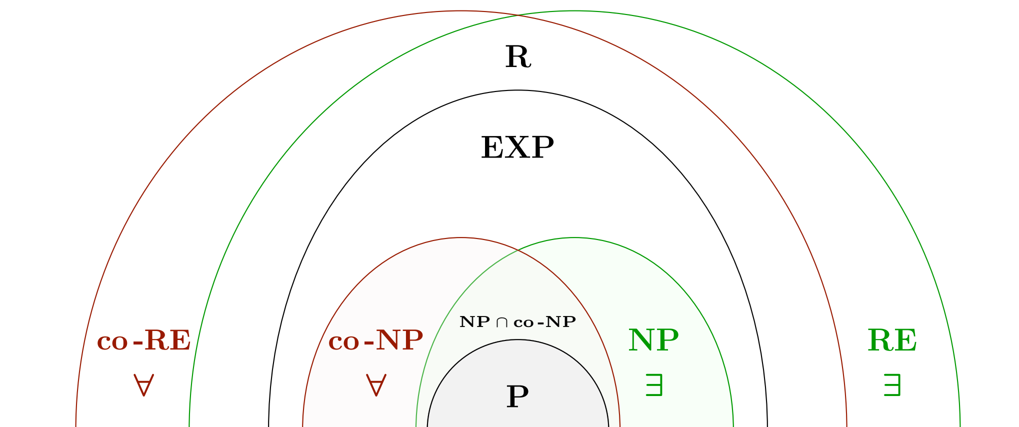
$$\texttt{UNSAT} = \{F \text{ in CNF} : \forall x\ F[x] = \texttt{False}\}.$$

- Boolean tautologies in DNF:

$$\texttt{TAUT} = \{F \text{ in DNF} : \forall x\ F[x] = \texttt{True}\}.$$

  It is easy to see that it is in **co-NP** by negating a formula in DNF and transferring the negation towards the variables using the de Morgan rule $\overline{\bigvee_i \bigwedge_j \ell_{ij}} \mapsto \bigwedge_i \bigvee_j \overline{\ell_{ij}}$, thus getting a formula in CNF with the opposite value.

The picture.



2 The deterministic time hierarchy

How do we know that more time allows us to decide more languages? In particular,

$$\mathbf{P} \subsetneq \mathbf{EXP} \subsetneq \mathbf{R}.$$

In the Automata course we have seen an application of the diagonal method to proving that $\mathbf{R} \subsetneq \mathbf{RE}$. Let us recall this proof and transform it into a proof of the deterministic time hierarchy.

The computability version. In the diagonal method we consider an infinite "matrix" with entries $M_i(M_j)$, where the lines (and rows) correspond to all possible machines $M_1, M_2, \ldots$ (or Turing numbers $1, 2, \ldots$). On the diagonal we have entries of the form $M(M)$, which gives the name to this method. The basic idea is to run M on its own encoding and to invert the answer.

Let us define the language

$$L = \{\langle M \rangle \ : \ M(\langle M \rangle) = 0\}.$$

It belongs to $\mathbf{RE}$, because one can run M and accept if it rejects; but if it does not stop, we don't stop either. (NB: In the actual lecture we used the language $M(\langle M \rangle) \neq 1$, which belongs to $\mathbf{co\text{-}RE}$.)

Does it belong to $\mathbf{R}$? No! Assume on the contrary that $L \in \mathbf{R}$. Consider the corresponding machine D that always stops. Is $\langle D \rangle \in L$? There are two cases:

1. $\langle D \rangle \in L$, thus D must accept $\langle D \rangle$. However, it means $D(\langle D \rangle) = 1 \neq 0$, and L is defined so that it means $\langle D \rangle \notin L$, a contradiction.

2. $\langle D \rangle \notin L$, thus D must reject $\langle D \rangle$ (because D always stops and thus must give some answer). However, it means $D(\langle D \rangle) = 0$, and L is defined so that it means $\langle D \rangle \in L$, a contradiction.

Thus both cases are impossible, and our assumption $L \in \mathbf{R}$ was wrong.

The complexity version. Let us consider a time-bounded version of this proof. For simplicity let us formulate it with specific constants and with time functions of the form n^k.

Let us define the language

$$L' = \{t \in \mathbb{N} \ : \ U \text{ rejects } \langle M_t, t \rangle \text{ in at most } |t|^{50} \text{ steps}\}.$$

(We could simply say that M_t rejects $\langle M_t \rangle$ in this number of steps, but slowing it down using a UTM will make our proof a little easier.) The constant 10 here serves for covering well the slowdown of the UTM we built (let us assume we built a quadratic-time UTM U, that is, the running time of U be bounded by $c' \cdot T^4$, where T is the number of steps of the machine it

simulates, and c' is a universal constant (note that the number of tapes and states is already accounted for in the length of the description of the simulated machine)).

This language is obviously in $\mathbf{DTime}[O(n^{50})]$, because U essentially decides it (it remains to flip the answer and take care of interrupting the machine after the specified number of steps).

Can one solve it much faster, say, in time $O(n^{10})$? Again, assume on the contrary that there is a machine D' that can do it that fast. Let d be a Turing number of D' that is large enough (recall that in our encoding every machine has infinitely many encodings, that is, Turing numbers). Namely, let the running time of D' be bounded by $c \cdot n^{10}$, let the running time of U be bounded by $c' \cdot T^4$ (if T is the number of steps of the machine it simulates), let b be the number of bits in d, and let b be so large that $\forall m > b\ c' \cdot (c \cdot m^{10})^4 < m^{50}$, that is, time b^{50} suffices to simulate D' using U on this length of the input. (This selection of d simply make the things as they should be eventually when one function grows asymptotically faster than another one. The machine M_d in the definition of L' is the machine D'.)

Is $d \in L$? Note that D' is fast and thus even U will make less than n^{50} steps when simulating it, so in the definition of L' the condition on the number of steps is satisfied. There are two cases:

1. $d \in L$, thus D' must accept d. However, it means $D'(d) = 1 \neq 0$, and L is defined so that it means $d \notin L$, a contradiction.

2. $d \notin L'$, thus D' must reject d. However, it means $D'(d) = 0$, and L is defined so that it means $d \in L$, a contradiction.

Thus both cases are impossible, and our assumption $L \in \mathbf{DTime}[O(n^{10})]$ was wrong.

Thus we proved the time hierarchy theorem for n^{10} vs n^{50}. One could formulate it for any "nice" time complexity function $f(n)$ satisfying the following three conditions.

Definition 2.1 (time-constructible function)**.** A function $f : \mathbb{N} \to \mathbb{N}$ is called time-constructible if

1. $\forall n \in \mathbb{N}\ f(n) \geqslant n$,

2. given 1^n (the unary representation of n) on the input, one can compute the binary representation of $f(n)$ in time $O(f(n))$.

3. f is non-decreasing.

Sometimes they omit the last condition, but it is convenient.

Why do we need time-constructibility? This is because one has to check that M_t stops in this number of steps, which means that one at least has to compute this number. Otherwise we might end up in a very strange situation where $\mathbf{DTime}[\xi(n)] = \mathbf{DTime}[2^{\xi(n)}]$ for a very weird function ξ — such (of course, hard-to-compute) functions indeed exist, but this is beyond the scope of this course.

Of course, all functions we typically use (like n^2, 2^n, or even 2^{2^n}) are time-constructible.

One can consider it an easy exercise (*now see its solution below*) to generalize our proof essentially replacing n^{10} by $f(n)$ and thus showing

Theorem 2.1. $\mathbf{DTime}[O(f(n))] \neq \mathbf{DTime}[O(f(n)^5))]$.

A much tighter theorem can be proved (with a little bit more than $f(n) \log f(n)$ instead of $f(n)^5$), but we will not prove it in this course (essentially, one needs to construct a $f(n) \log f(n)$-time Universal Turing Machine).

As a corollary, one can see

$$\mathbf{P} \neq \mathbf{EXP} \neq \mathbf{EEXP}.$$

This is the main takeout of this lesson.

2.1 A more formal treatment of the DTime hierarchy

Let us do that "exercise". We start with formulating what we need from our universal Turing machine that we built in the Automata course. We won't use the tightest possible bound, but we will use a formulation with universal constants not depending on the machine being simulated. We can do that because if you remember how we built a UTM, things that depend on the description of the input machine were clearly polynomial-time (in the size of that description).

Lemma 2.1 (UTM, from the Automata course)**.** *There exist a DTM U and a polynomial p such that for every DTM M and x*

$$U(\langle M, x\rangle) = M(x),$$

moreover, U stops on input $\langle M, x\rangle$ in at most $p(n)$ steps, where $n = \max(|\langle M, x\rangle|, time_M(x))$.

This polynomial $p(n)$ is in fact a very small specific polynomial, but we do not care about it now. Note that the coefficient of p (including the constant factor — we did not write $O(\ldots)$ separately) do not depend on M (and, of course, on x).

Theorem 2.2. *Let $f : \mathbb{N} \to \mathbb{N}$ be time-constructible. There is a polynomial q such that*

$$\mathbf{DTime}[f(n)] \subsetneq \mathbf{DTime}[q(f(3n+1))].$$

Proof. Consider the following function:

$$H(\langle M, x\rangle) = \text{“if } M(x) \text{ accepts in at most } f(|x|)^2 \text{ steps, then reject; otherwise accept”}.$$

It can be implemented using the UTM from Lemma 2.1 and an alarm clock in time $q(f(n))$, where q is a polynomial and $n = |\langle M, x\rangle|$ (recall that $f(n) \geqslant n$ and the value of $f(n)$ for the alarm clock can be computed in time $O(f(n))$ by Def. 2.1; squaring is already taken into account in the definition of q, which may be larger than the polynomial in the lemma).

Let us build a new function

$$D(\langle M\rangle) = H(\langle M, M\rangle).$$

It can be implemented (using some DTM M_D) in time $q(f(3n+1))$, where now $n = |\langle M\rangle|$ and $3n+1$ stands for the size of the encoding of the pair (M, M) (we can do it more efficiently, but $3n+1$ is certainly enough: encode a pair (a, b) as $1^{|a|}0ab$, then everyone can easily extract a and b from it).

Thus $L(M_D) \in \mathbf{DTime}[q(f(3n+1))]$. Let us prove that $L(M_D) \notin \mathbf{DTime}[f(n)]$, which will complete the proof of our theorem.

Assume for the sake of contradiction that $L(M_D) \in \mathbf{DTime}[f(n)]$, in particular, let C be a DTM running in time at most $c \cdot f(n)$, where c is a constant, and let $L(M_D) = L(C)$. Since $f(n)^2$ grows asymptotically faster than $c \cdot f(n)$, let N be so large that $\forall n \geqslant N \; c \cdot f(n) < f(n)^2$.

Consider C_* that has absolutely the same program (and running time!) as C, but its description size n_* is more than N (recall that we assumed that in our encoding every machine appears infinitely many times — one can always append some garbage at the end of the description). Then the time bound in the definition of $H(\langle C_*, C_*\rangle)$ guarantees that C_* stops before the alarm clock rings (namely, $C_*(\langle C_*\rangle)$ stops in time $c \cdot f(n_*)$ and the time bound in $H(\langle C_*, C_*\rangle)$ is $f(|\langle C_*\rangle|)^2 = f(n_*)^2$) and thus the question in the definition of H only concerns acceptance/rejectance and not the running time bound, that is,

$$D(\langle C_*\rangle) = 1 \iff H(\langle C_*, C_*\rangle) = 1 \iff C_*(\langle C_*\rangle) = 0. \tag{2}$$

It gives us a contradiction similarly to the diagonalization argument in the proof of $\mathbf{R} \neq \mathbf{RE}$: since we assumed that C (and C_*) decides $L(D)$: consider the two possibilities for the answer of $C_*(\langle C_*\rangle)$,

- if $C_*(\langle C_*\rangle) = 0$, then $D(\langle C_*, C_*\rangle) = 0$,
- if $C_*(\langle C_*\rangle) = 1$, then $D(\langle C_*, C_*\rangle) = 1$,

and both possibilities contradict (2). □

Corollary 2.1. $\mathbf{P} \neq \mathbf{EXP} \neq \mathbf{EEXP}$.

Proof. Choose some function (such as $2^{\sqrt{n}}$) in between all the polynomials and some exponent and use Theorem 2.2 for this function: $\mathbf{P} \subseteq \mathbf{DTime}[2^{\sqrt{n}}] \neq \mathbf{DTime}[2^{q(3\sqrt{n}+1)}] \subseteq \mathbf{EXP}$. Then do the same between all the exponents and a doubly exponential function. □

Remark 2.1. By analogy, one can prove the deterministic space hierarchy theorem. The difference in the definition of $H(\langle M, x\rangle)$ will be that the restriction will be on space (and not on time) spent while simulating M. Since one can desing a UTM that essentially does not spend extra space (simulates a space-$S(n)$ using space $O(S(n))$, the hierarchy will be even more tight: we won't need q, we just need the function bounding the space for the smaller class $s(n)$ to grow asymptotically slower than the function bounded the space for the larger class $S(n)$, i.e., $s(n) = o(S(n))$. Everything else will be the same. It will work for *space-computable* functions, that is, non-decreasing, at least $\log n$ (otherwise we won't be able even to keep n), and computable within the same space (that is, $S(n)$ can be computed withing space $O(S(n))$). Completing this proof of $\mathbf{DSpace}[s(n)] \neq \mathbf{DSpace}[S(n)]$ is an easy exercise.

3 Reductions and completeness

In this course we will see several types of reductions, and there are many more of them. A problem W is reducible to a problem S if a device that solves S can help us solving problem W as well, that is, there is a machine that reduces the question about W to the question about S. This is an informal definition: Several formal instantiations of it are on the way. We will denote reducibilities as $W \leqslant S$ with various subscripts and/or superscripts over $\leqslant$, as they define a partial order on problems; sometimes we denote them by $W \to S$ as well for the purpose of illustration.

Even though we have not yet defined formally what does it mean to be reducible, we can already define the notions of hardness and completeness (that do depend on the types of reductions!).

Definition 3.1. A problem H is called hard for a class $\mathcal{C}$ ($\mathcal{C}$-hard) if for every problem $W \in \mathcal{C}$, there is a reduction of W to H.

A problem H is called complete for a class $\mathcal{C}$ ($\mathcal{C}$-complete) if it is hard for $\mathcal{C}$ and it also belongs to $\mathcal{C}$.

Note that a $\mathcal{C}$-hard problem can be outside $\mathcal{C}$, in particular, it can be much harder!

It is important to note that this definition depends on the type of reductions, so in full it would sound as "H is complete for $\mathcal{C}$ under [such and such type of] reductions". Sometimes the type of reductions is clear from the context, sometimes it is stated explicitly.

3.1 Many-one reductions (without a time bound), RE-completeness, negative results

Many-one reductions. Let us start with the simplest type of reduction: a many-one reduction for languages (decision problems).

Definition 3.2. A language W is many-one reducible to a language S (we denote it simply $W \leqslant S$) if there is a function $f : \{0,1\}^* \to \{0,1\}^*$ computable by a DTM (that, of course, always stops and outputs its result) such that $\forall x \in \{0,1\}^*\ x \in W \iff f(x) \in S$.

This function is called a many-one reduction of W to S.

Given such a function it is easy to solve W given a device for S: One simply applies f to the input x and returns the answer that this device returns for $f(x)$.

Example 3.1. The acceptance problem (does M accept x?):

$$\mathtt{AP} = \{\langle M, x\rangle \ :\ M(x) = 1\}.$$

The halting problem (does M stops on x?):

$$\mathtt{HP} = \{\langle M, x\rangle \ :\ M(x) \in \{0,1\}\}.$$

A many-one reduction from $\mathtt{AP}$ to $\mathtt{HP}$:

$$f(\langle M, x\rangle) = \langle M', x\rangle,$$

where M' is the machine M with its rejecting state replaced with an infinite loop. (Of course, it is easy to make this change in the program of M, and there is a DTM that will do that for us and return $\langle M', x\rangle$ on the input $\langle M, x\rangle$.)

Indeed, if $x \in \mathtt{AP}$, then M accepts x, and M' still accepts x (and thus stops), so $\langle M', x\rangle \in \mathtt{HP}$. If $x \notin \mathtt{AP}$, then M either rejects x or does not stop, in both cases M' does not stop, so $\langle M'\rangle \notin \mathtt{HP}$.

Properties of many-one reductions. Many-one reducibilities are reflexive ($W \leqslant W$) and transitive ($A \leqslant B \leqslant C$ implies $A \leqslant C$), the latter is clear from using one reduction as a subprogram inside the other reduction: the reduction of A to C is a composition of reductions of A to B (call it f_1) and B to C (call it f_2): $f(x) = f_2(f_1(x))$.

They are not necessarily symmetric: it may (or may not) happen that $A \leqslant B$ but $B \nleqslant A$.

Note, however, that a many-one reduction of W to S also serves as a many-one reduction from $\overline{W}$ to $\overline{S}$ with absolutely no change, this is the same function.

Our classes $\mathbf{R}, \mathbf{RE}, \mathbf{co\text{-}RE}$ are closed under these reductions:

Lemma 3.1.
- *If $S \in \mathbf{R}$ and $W \leqslant S$, then $W \in \mathbf{R}$,*
- *If $S \in \mathbf{RE}$ and $W \leqslant S$, then $W \in \mathbf{RE}$,*
- *If $S \in \mathbf{co\text{-}RE}$ and $W \leqslant S$, then $W \in \mathbf{co\text{-}RE}$.*

The proof follows by composing a DTM for the reduction with the appropriate machine for S.

Completeness and hardness under many-one reductions. According to Def. 3.1, an **RE**-hard language is a language such that every language in **RE** is reducible to it (and if it's in **RE** itself, it's called **RE**-complete). A **co-RE**-hard (resp., **co-RE**-complete) langauge is defined similarly. By default, for computability classes (**RE**, **co-RE**, **R**) we will mean many-one reductions. Let us prove that an **RE**-complete language exists.

Theorem 3.1. $\mathtt{AP}$ *is* **RE**-*complete.*

Proof. $\mathtt{AP} \in \mathbf{RE}$ because the universal Turing machine accepts exactly $\mathtt{AP}$.

Consider a language $W \in \mathbf{RE}$, let us reduce it to $\mathtt{AP}$. Since W is recursively enumerable, there is a DTM M_W that accepts W. Here is a reduction:

$$f(x) = \langle M_W, x\rangle.$$

The DTM that computes f, given x, simply writes down the description of M_W and x (arranging them in the encoding of a pair of strings). Note that the whole program of M_W is embedded in this DTM as a sequence of states that serve the only purpose of writing down this fixed string $\langle M_W\rangle$ no matter what. Obviously $x \in W \iff f(x)(= \langle M_W, x\rangle) \in \mathtt{AP}$ according to the definition of $\mathtt{AP}$. □

Once we have one **RE**-complete problem, we have more of them (and of **co -RE**-complete problems as well): the following theorem is obvious from the definitions.

Theorem 3.2. *1. If H is* **RE***-hard and $H \leqslant S$, then S is* **RE***-hard.*

2. If H is **co -RE***-hard, then $\overline{H}$ is* **co -RE***-hard.*

Corollary 3.1. `HP` *is* **RE***-complete.*

Proof. See a reduction in Example 3.1 and use the **RE**-completeness of `AP`. Do not forget to demonstrate als othat `HP` $\in$ **RE** itself, but this is obvious: a DTM can simulate $M(x)$ using DTM and accept if M stops (irrespectively of its 0/1 output). □

Applications to negative results. Many-one reductions are useful for proving that some language is not recursive (and sometimes even not recursively enumerable!): we use the fact that our classes are closed under these reductions.

Example 3.2. We proved above that the language

$$L = \{\langle M\rangle \;:\; M(\langle M\rangle) = 0\}$$

is not recursive. In the lecture we proved this (with the same proof) for the language

$$L' = \{\langle M\rangle \;:\; M(\langle M\rangle) \neq 1\}.$$

Let us reduce these languages to `AP` or to $\overline{\texttt{AP}}$, this will imply that `AP` is not recursive as well (if it would be recursive, then L would be recursive as well!). One proof would be enough, of course. Let us start with $L' \leqslant \overline{\texttt{AP}}$. Consider

$$f(\langle M\rangle) = \langle M, M\rangle.$$

If $\langle M\rangle \in L'$, then $M(\langle M\rangle) \neq 1$, this $\langle M, M\rangle \in \overline{\texttt{AP}}$. If $\langle M\rangle \notin L'$, then $M(\langle M\rangle) = 1$, this $\langle M, M\rangle \notin \overline{\texttt{AP}}$.

We can do it similarly for $L \leqslant \texttt{AP}$.

$$f(\langle M\rangle) = \langle \overline{M}, M\rangle,$$

where $\overline{M}$ is the same as M, but the accepting and rejecting states are interchanged. If $\langle M\rangle \in L$, then $M(\langle M\rangle) = 0$, so $\overline{M}(\langle M\rangle) = 1$, and thus $\langle \overline{M}, M\rangle \in \texttt{AP}$. If $\langle M\rangle \notin L$, then $M(\langle M\rangle) \neq 0$, so $\overline{M}(\langle M\rangle) \neq 1$, and thus $\langle \overline{M}, M\rangle \notin \texttt{AP}$.

Let us formulate this example as a theorem (that we actually proved in the Automata course):

Theorem 3.3. `AP` $\in$ **RE** $\setminus$ **R**.

Remark 3.1. What happens if an **RE**-hard problem is in **co-RE**? It simply does not happen: if so, all **RE** problems are in **co-RE**, and by symmetry **RE** = **co-RE**, and we know that $\mathbf{R} = \mathbf{RE} \cap \mathbf{co\text{-}RE} (= \mathbf{RE}$ now), a contradiction with $\mathbf{RE} \neq \mathbf{R}$.

From the Automata course we know that there are languages outside $\mathbf{RE} \cup \mathbf{co\text{-}RE}$: this follows by a counting argument, the cardinality of this class is $\aleph_0$ because there are only that many Turing machines, while the cardinality of the class of all possible languages is $2^{\aleph_0}$, much larger. As another application of reductions let us give a constructive proof.

Example 3.3. Let us prove that the following language is outside $\mathbf{RE} \cup \mathbf{co\text{-}RE}$:

$$L_\infty = \{\langle M \rangle \; : \; |L(M)| = \infty\}.$$

We will do it by demonstrating both its **RE**-hardness and **co-RE**-hardness.

Indeed, HP can be reduced to it using the following reduction

$$f(\langle M, x \rangle) = \langle M_x \rangle,$$

where M_x is the machine that on input y works as follows:

1. Simulate $|y|$ steps of $M(x)$ (using UTM).
2. If M stops within this time period, then accept.
3. Otherwise reject.

If $\langle M, x \rangle \in$ HP, that is, M stops on x, there is a specific number of steps t such that for every y of length at least t our machine M_x will accept. That is, M_x accepts an infinite amount of strings.

Otherwise M_x rejects all the strings.

Also $\overline{\text{HP}}$ can be reduced to L_∞ using the following reduction

$$f(\langle M, x \rangle) = \langle S_x \rangle,$$

where M_x is the machine that on input y works as follows:

1. Simulate $|y|$ steps of $M(x)$ (using UTM).
2. If M stops within this time period, then reject.
3. Otherwise accept.

If $\langle M, x \rangle \in \overline{\text{HP}}$, that is, M does not stops on x at all, then S_x accepts everything, $L(S_x) = \{0, 1\}^*$, it is infinite.

Otherwise M stops on x, and there is a specific number of steps t such that for every y of length at least t our machine S_x will reject. That is, S_x accepts at most $\{0, 1\}^t$ strings, that is, $L(S_x)$ is finite.

Since these two reductions make L_∞ both **RE**-hard and **co-RE**-hard, it cannot belong neither to **co-RE** nor to **RE**.

Example 3.4. Consider the following language:

$$L_{all} = \{\langle M \rangle \ : \ M \text{ is a DTM that accepts every binary string}\}.$$

Let us prove that $L_{all} \notin \mathbf{RE} \cup \mathbf{co\text{-}RE}$ with a single many-one reduction to it, that is, by proving that $L_\infty \leqslant L_{all}$ (and thus the result follows because both **RE** and **co-RE** are closed under many-one reductions). Here it is:

$$f(\langle M \rangle) = \langle M' \rangle,$$

where M' on input y works as follows. It runs "in parallel"[1] the machine M on every possible input; when a new input is accepted in one of the parallel processes[2], we increase a counter of accepted inputs. Our machine M' accepts its input y if the counter comes to the number of y in the length-increasing lexicographic order, that is, the counter comes to the number with the binary description $1y$.

If $\langle M \rangle \in L_\infty$, then M' will eventually increase its counter to every natural number and thus it will accept every possible y, i.e., $\langle M' \rangle \in L_{all}$.

If $\langle M \rangle \notin L_\infty$, then the counter will stop at some moment and will never increase again, thus all y's that are greater than that won't be accepted, i.e., $\langle M' \rangle \notin L_{all}$.

3.2 Polynomial-time many-one reductions and NP-completeness

Almost all properties of many-one reductions goes through for their polynomial-time bounded version. Yet where we come to a contradiction, in the polynomial-bounded case we typically come to open questions.

Many-one polynomial-time reductions.

Definition 3.3. A language W is polynomial-time many-one reducible to a language S (we denote it simply $W \leqslant^p S$) if there is a function $f : \{0,1\}^* \to \{0,1\}^*$ computable by a DTM that takes polynomial time such that $\forall x \in \{0,1\}^* \ x \in W \iff f(x) \in S$.

This function is called a many-one polynomial-time reduction of W to S.

An example (`Circuit-SAT` $\leqslant^p$ `3-SAT`) is delayed to the next lecture, but you have certainly seen many examples of such reductions in the Algorithms-2 course.

[1] Recall this construction from the Automata course: in order to run many processes in parallel on a single Turing machine so that every process eventually finishes (if it would stop when run alone), enumerate the "stages" of our machine as $u(n, k)$, where n is the input viewed as a natural number by prepending 1 to it, and k is the number of steps. At each stage $\ell = u(n, k)$ our machine extracts n and k from ℓ and runs (even from the scratch!) the machine we want on input n for k steps.

[2] We count it only once if the machine has already stopped before.

Properties of many-one polynomial-time reductions. Simularly to their unbounded version, many-one polynomial-time reducibilities are reflexive ($W \leqslant^p W$) and transitive ($A \leqslant^p B \leqslant^p C$ implies $A \leqslant^p C$), the latter is clear from using one reduction as a subprogram inside the other reduction: the reduction of A to C is a composition of reductions of A to B (call it f_1) and B to C (call it f_2): $f(x) = f_2(f_1(x))$. To check it works in polynomial time note that the size of $f_1(x)$ is bounded by a polynomial in $|x|$ (the polynomial-time machine for this reduction simply has no time to write more bits). Thus the running time of f_2 is a polynomial of a polynomial of $|x|$ and overall we can compute f in time $q_1(n) + q_2(q_1(n))$, where q_1 and q_2 are the polynomials bounding the running time of the machines for f_1 and f_2, respectively, and $n = |x|$.

The classes $\mathbf{P}, \mathbf{NP}, \mathbf{co\text{-}NP}$ are closed under these reductions:

Lemma 3.2.
- *If $S \in \mathbf{P}$ and $W \leqslant^p S$, then $W \in \mathbf{P}$,*
- *If $S \in \mathbf{NP}$ and $W \leqslant^p S$, then $W \in \mathbf{NP}$,*
- *If $S \in \mathbf{co\text{-}NP}$ and $W \leqslant^p S$, then $W \in \mathbf{co\text{-}NP}$.*

The proof follows by composing a DTM for the reduction with the appropriate machine for S.

Completeness and hardness under many-one reductions. According to Def. 3.1, an **NP**-hard language is a language such that every language in **NP** is reducible to it using a polynomial-time many-one reduction (and if it's in **NP** itself, it's called **NP**-complete). A **co-NP**-hard (resp., **co-NP**-complete) language is defined similarly. By default, for time-bounded complexity classes (**NP**, **co-NP**, **P**) we will mean many-one polynomial-time reductions (however, other classes, like space-bounded classes or parallel complexity classes require different types of reductions). We will later prove that an **NP**-complete language exists.

Applications to negative results. While for the computability classes we have seen examples of negative results as consequences of reducibilities, in the complexity case we usually arrive at open questions. We will later learn more about possible consequences, let us now make just a single remark.

Remark 3.2. What happens if an **NP**-hard problem is in **co-NP**? If so, all **NP** problems are in **co-NP**, and by symmetry **NP** = **co-NP**. It would be a great progess: we still do not know whether **NP** = **co-NP**... (but there is no contradiction... so far).

Exercise 3.1. Warning! If one applies a different type of reductions the results may be different, for example, the closure properties may not hold or the reductions will be not useful by another reason, or complete problems will stop being complete, or there may be even no complete problems at all.

As an exercise, think about many-one reductions (and completeness) for **P** and **NP**, and about many-one polynomial-time reductions (and completeness) for **R** and **RE**. Think

also about two problems that are many-one reducible to each other but not polynomial-time many-one reducible to each other.

An example: a reduction from `Circuit-SAT` to `3-SAT`

Lemma 3.3 (Tseitin's reduction). `Circuit-SAT` *is polynomial-time many-one reducible to* `3-SAT`.

Proof. Given a circuit C with inputs $x_1, x_2, \ldots, x_n$ and $m - n$ internal gates, our reduction will output a formula F in 3-CNF in variables $y_1, y_2, \ldots, y_m$ such that

$$\exists a_1, \ldots, a_n \in \{0,1\} \; C(a_1, \ldots, a_n) = 1 \iff \exists b_1, \ldots, b_m \in \{0,1\} \; F(b_1, \ldots, b_m) = 1.$$

Even more than that, we will prove that

$$\forall a_1, \ldots, a_n \in \{0,1\} \; (C(a_1, \ldots, a_n) = 1 \Leftrightarrow \exists b_{n+1}, \ldots, b_m \in \{0,1\} \; F(a_1, \ldots, a_n, b_{n+1}, \ldots, b_m) = 1). \tag{3}$$

In other words, every satisfying assignment for C can be extended to a satisfying assignment assignment to F.

We identify x_ℓ with y_ℓ for $\ell \leqslant n$. Let us enumerate the internal gates of C in a topological order starting from $n + 1$, thus a variable y_k will be associated with an input or an internal gate number k, and y_m will be associated with the output gate.

For every internal gate number k of C computing a binary[3] function $f(y_i, y_j)$ of its inputs with numbers $i, j < k$, our formula F will include all clauses[4] in variables y_i, y_j, y_k that yield $f(y_i, y_j) = y_k$ (the correct computation of y_k from y_i, y_j). Let us denote the conjunction of all these clauses by F_k. Note that

$$f(b_i, b_j) = b_k \iff F_k(b_i, b_j, b_k) = 1 \tag{4}$$

The conjunction of all formulas F_k for all the internal gates of C says yes if the values of y_k's are computed correctly from the inputs. It remains to add a clause (y_m) stating that the value computed in the output is 1. Therefore, our reduction outputs the formula F defined as

$$F = (y_m) \wedge \bigwedge_{k=n+1}^{m} F_k.$$

Given a description of C, it is easy to write down all these clauses, so this reduction works in polynomial time.

On the other hand, (4) guarantees that the values of a satisfying assignment for $x_1, \ldots, x_n$ can be extended by the values of $y_{n+1}, \ldots, y_m$ so that the resulting assignment satisfies F. In the other direction if there is a satisfying assignment for F, then dropping the values of $y_{n+1}, \ldots, y_m$ we obtain a satisfying assignment for C. Thus we get (3), so the reduction is correct. ☐

[3] The case of a unary function is left as an easy exercise.

[4] A useful exercise: write down these clauses, for example, for the case $f(y_i, y_j) = y_i \oplus y_j$.

The existence of NP-complete problems. Later we will prove the Cook-Levin theorem (3-SAT is **NP**-complete). However, for now let us prove that at least some **NP**-complete problem exists.

Definition 3.4 (BH)**.** The following problem is commonly called Bounded Halting even though it would be better to call it Nondeterministic Bounded Acceptance:

$$\texttt{BH} = \{\langle N, x, 1^t \rangle : N \text{ is a NTM, } N \text{ accepts } x \text{ in at most } t \text{ steps}\}.$$

Here by "accepts x in at most t steps" we mean that there **exists** a sequence of nondeterministic choices that drives $N(x)$ into the accepting state. (We do not care about other sequences of nondeterministic choices.)

Lemma 3.4. BH *is* **NP**-*complete.*

Proof. BH $\in$ **NP** because we can simulate N for t steps nondeterministically. We did not prove that there is a UTM for NTMs, but we can easily construct it by analogy with a UTM for DTMs (just making a nondeterministic choice for the next instruction), or alternatively by converting N to a "DTM with a witness", selecting this witness (of length bounded by t) nondeterministically and then using our UTM for DTMs.

Let us prove that BH is **NP**-hard. Consider a language $L \in$ **NP** and let us prove that $L \leqslant^p$ BH. Let N be a polynomial-time NTM defining L, and let p be a polynomial bounding its running time. Our reduction is

$$f(x) = \langle N, x, 1^{p(|x|)} \rangle.$$

Given x, it is easy to write down this triple (we just need to compute $p(|x|)$), thus it works in polynomial time. Also

$$x \in L \iff N \text{ accepts } x \iff f(x) \in \texttt{BH},$$

because "accepts" and "accepts in $p(|x|)$ steps" is the same thing for N since its running time is bounded by p. □

With Lemmas 3.4, 3.3 in hand, it remains to reduce BH to Circuit-SAT in order to prove the Cook–Levin theorem.

3.3 Other types of reductions

We now define a much more general type of reductions that allow to use their "black box" in an arbitrary way. (Many-one reductions use it only once and give the same answer.) In programming terms, an oracle machine uses its oracle as an external subroutine.

For this purpose, let us define a variant of Turing machines called oracle Turing machines. These are machines that are given access to an "oracle" that can magically (and instantly) solve a certain problem. Let us start with reductions between languages, then an oracle is a language, that is, a function $\{0,1\}^* \to \{0,1\}$.

Definition 3.5. (Oracle Turing machines) An oracle Turing machine is a DTM $M^\bullet$ that has a special tape (an "oracle tape") and special oracle states q_{query}, $q_{answer,yes}$, $q_{answer,no}$. The bullet $\bullet$ over it says that it has a "hole" (or an "interface") where we can insert any oracle (any language, that is, any function $\{0,1\}^* \to \{0,1\}$).

To execute $M^\bullet$, one needs to use a specific oracle language $O \subseteq \{0,1\}^*$ (the resulting construction — an oracle machine using a specific oracle — is denoted M^O). Whenever during the execution M enters the state q_{query}, the oracle solves the problem in a single step and the machine transits to the corresponding state $q_{answer,yes}$ or $q_{answer,no}$.

We can now define Turing reductions (also called oracle reductions).

Definition 3.6 (Turing reductions)**.** We say that a language W is Turing-reducible to a language S (denoted $W \leqslant_T S$) if there is an oracle DTM $R^\bullet$ that stops on every input[5] and such that

$$\forall x \in \{0,1\}^* \ (x \in W \iff R^S(x) = 1).$$

Remark 3.3. Note that if a language is Turing-reducible to S, then it is also Turing-reducible to $\overline{S}$, because one can use the latter oracle and then invert its yes/no answer to get the answer of the former oracle.

This is very different from many-one reductions, as HP is not reducible to $\overline{\texttt{HP}}$!

Later in Section 9.2 we will generalize Turing reductions to other types of problems and will consider their polynomial-time version as well (Definition 9.3).

3.4 Padding

A small note about how the complexity of a language can be easily lowered. Consider a language $L \in \mathbf{DTime}[t(p(n))]$ for time-computable functions t and p, let M be the DTM deciding L in time $O(t(p(n)))$. Then consider a padded language

$$L_{pad} = \{x01^k : x \in L,\ k = p(|x|) - |x| - 1\}$$

for some easily computable function p. That is, we add a clearly distinguishable "tail" to the words of L. Then $L_{pad} \in \mathbf{DTime}[t(n)]$. Indeed, to decide $y \in L_{pad}$, we simply check that the tail is there (reading y from the end), thus figure out was is x, compute $p(|x|)$ and compare it to the input length, and if these checks are OK, run the $O(t(p(n)))$ algorithm on x. The running time is $O(t(p(|x|)))$, but the input length is $|y| = p(|x|)$, thus the running time is $O(t(|y|))$.

Here is an example of the use of this technique.

[5] And for every possible oracle.

Proposition 3.1. *If* $\mathbf{P} = \mathbf{NP}$*, then* $\mathbf{EXP} = \mathbf{NEXP}$*.*

Proof. Consider $L \in \mathbf{NEXP}$, let us show that $L \in \mathbf{EXP}$. Consider a NTM running in time at most $2^{q(n)}$, for a polynomial $q(n)$, and deciding L. Consider L_{pad} as above for $t(n) = n$, $p(n) = 2^{q(n)}$. This language is in $\mathbf{NP}$, thus by our assumption it is in $\mathbf{P}$ and there is a deterministic $d(n)$-time algorithm D for it, where d is a polynomial. But L then can be decided deterministically by the following procedure:

- On input x of length n, pad it to y of length $2^{q(n)}$.
- Run $D(y)$ and return its answer.

Since D runs in time $d(|y|)$, the resulting algorithm runs in time $O(d(2^{q(n)}))$, which is $O(2^{q'(n)})$ for a polynomial q', i.e., $L \in \mathbf{EXP}$. □

4 Rice's theorem and more undecidabile languages

Let us consider properties (that is, sets) of languages, such as "a language is empty", "a language is infinite", etc. Rice's theorem says that every nontrivial property of **RE** languages is undecidable (the input to this problem is the description of a DTM accepting the language, certainly there is at least one such machine for every language in **RE**).

Theorem 4.1. *Let* $\Pi \subseteq \mathbf{RE}$*,* $\Pi \neq \emptyset$*,* $\Pi \neq \mathbf{RE}$*. Then the language*

$$L_\Pi = \{\langle M \rangle \ : \ L(M) \in \Pi\}$$

is undecidable (here M *denotes a DTM).*

Proof. Let us consider two cases. In the first case assume that $\emptyset \notin \Pi$ (please note that here $\emptyset$ is a language while in the statment of the theorem $\emptyset$ is a set of languages!). Let D be some DTM such that $L(D) \in \Pi$ — there must be at least one (r.e.) language in Π, thus D exists.

Let us construct a many-one reduction from `HP` to L_Π (which will prove the undecidability result in our first case). Given $\langle M, x\rangle$, our reduction will output the description of the following machine M'_x working on input y:
— Write down x and proceed as M on x. If $M(x)$ stops, then proceed as $D(y)$.

Note that if M stops on x, then M'_x accepts the same language as D does! That is, this reduction correctly maps $\langle M, x\rangle \in$ `HP` to a machine M'_x such that $\langle M'_x\rangle \in L_\Pi$.

Otherwise it does not accept any word at all, that is, our reduction correctly maps $\langle M, x\rangle \notin$ `HP` to our machine M'_x accepting the empty language, that is, to a machine M'_x such that $\langle M'_x\rangle \notin L_\Pi$ (recall the condition of our first case).

The second case is similar: if $\emptyset \in \Pi$, let us consider $\Pi' = \mathbf{RE} \setminus \Pi$. It still satisfies the condition of the theorem, and also of the first case. On the other hand, $L_\Pi \in \mathbf{R} \iff L_{\Pi'} \in \mathbf{R}$ because $L_\Pi = \overline{L'_\Pi}$ (if we assume some encoding such that every string encodes some machine — we call one of such encodings "Turing numbers"). □

The following straightforward corollary of the proof has been formulated in problem sessions:

Corollary 4.1 ("Extended Rice's theorem")**.** *Under the conditions of the theorem, (1) if additionally $\emptyset \notin \Pi$, then $L_\Pi \notin \mathbf{co\text{-}RE}$; (2) if additionally $\emptyset \in \Pi$, then $L_\Pi \notin \mathbf{RE}$.*

Proof. (1) In the first part of the proof of the theorem we reduce `HP` to L_Π, thus L_Π is **RE**-hard and cannot belong to **co-RE**. (2) In the second part we consider $\Pi' = \mathbf{RE} \setminus \Pi$, thus $\emptyset \notin \Pi'$ and we can use the first part of the theorem for it. That is, $L_{\Pi'} \notin \mathbf{co\text{-}RE}$, and it means $L_\Pi \notin \mathbf{RE}$. □

Let us consider several examples where we can and where we cannot use Rice's theorem to prove undecidability results. In all these examples M denotes a DTM.

Example 4.1. Consider

$$L_{\leqslant 3} = \{\langle M \rangle \ : \ M \text{ accepts at most three words}\}.$$

Although the corresponding property is not a subset of **RE**, one can consider the complement of this language (recall that the complement is undecidable if and only if the language is undecidable), namely,

$$L_{\geqslant 4} = \{\langle M \rangle \ : \ M \text{ accepts at least four words}\}.$$

Now this is a property of r.e. languages, it is nontrivial (there are r.e. languages that contain at most three words as well as languages that contain at least four words), thus one can apply Rice's theorem to it and conclude that $L_{\geqslant 4}$ (and thus $L_{\leqslant 3}$) is undecidable.

Example 4.2. Consider

$$L_{\mathbf{co\text{-}RE}} = \{\langle M \rangle \ : \ M \text{ accepts a language in } \mathbf{co\text{-}RE}\}.$$

While this language looks a bit overcomplicated, note that every DTM accepts a language in **RE** (by definition), thus the condition requires it to accept a language in $\mathbf{RE} \cap \mathbf{co\text{-}RE} = \mathbf{R}$. That is, the property is the set of recursive languages (**R**), and it satisfies the condition of Rice's theorem ($\mathbf{R} \subseteq \mathbf{RE}$, $\mathbf{R} \neq \emptyset$, $\mathbf{R} \neq \mathbf{RE}$).

Example 4.3. Consider

$$L_0 = \{\langle M \rangle \ : \ M(\varepsilon) = 0\}.$$

Note that we *cannot* apply Rice's theorem directly to L_0, because this condition is *not a property of a language*, it is rather a property of the machine M. (There may be two machines for the same language L, one rejects ε and the other one does not stop on it, the result is the same — the language is the same, but one machine belongs to L_0 and the other one does not.)

It does *not* mean that the language is decidable — we simply cannot apply Rice's theorem to it, so we have to prove the undecidability result differently (actually we did it before).

Example 4.4. Consider

$$L_1 = \{\langle M \rangle \,:\, M(\varepsilon) \neq 1\}.$$

Yes we can apply Rice's theorem to it: $\varepsilon \notin L(M) \iff M(\varepsilon) \neq 1$, thus we can reformulate it as a property of (r.e.) languages, not machines:

$$L_1 = \{\langle M \rangle \,:\, \varepsilon \notin L(M)\}.$$

It is nontrivial, thus this language is undecidable.

5 The Arithmetical Hierarchy

Let us define a hierarchy of classes of languages that are harder and harder to compute. We will give two definitions for this hierarchy and then prove their equivalence.

Before doing that, let us define classes of languages with oracle access to other languages.

5.1 Classes defined using oracles

Informally, for two classes $\mathcal{C}$ and $\mathcal{D}$, the class $\mathcal{C}^{\mathcal{D}}$ contains languages accepted by machines defining the class $\mathcal{C}$ with oracle access to languages in the class $\mathcal{D}$. However, there are nuances in this definition: for example, there are several equivalent definitions of **NP** that use different types of machines (DTMs or NTMs), so which one would we mean in $\mathbf{NP}^{\mathcal{D}}$? Therefore, it is safer to define specific classes explicitly.

Definition 5.1. For two languages L, B, we say that L is recursive in B if there is an oracle DTM $M^\bullet$ that always stops such that $L = L(M^B)$.

In other words, L is Turing-reducible to B.

For a class $\mathcal{D}$, we define

$$\mathbf{R}^{\mathcal{D}} = \{L \subseteq \{0,1\}^* :\ \text{there is } B \in \mathcal{D} \text{ such that } L \text{ is Turing-reducible to } B\}.$$

Remark 5.1. Note that we always give access to a single oracle. If $\mathcal{D}$ has complete problems, it is not a problem: if you want to use several oracles in this class, just use a single $\mathcal{D}$-complete oracle H after reducing your queries to H.

Definition 5.2. For two languages L, B, we say that L is recursively enumerable in B (r.e. in B) if there is an oracle DTM $M^\bullet$ such that $L = L(M^B)$.

For a class $\mathcal{D}$, we define

$$\mathbf{RE}^{\mathcal{D}} = \{L \subseteq \{0,1\}^* :\ \text{there is } B \in \mathcal{D} \text{ such that } L \text{ is r.e. in } B\}.$$

5.2 The oracle-style definition of AH

The first level of the Arithmetical Hierarchy is formed by the classes we already know:

$$\begin{aligned} \Delta_1^0 &= \mathbf{R}, \\ \Sigma_1^0 &= \mathbf{RE}, \\ \Pi_1^0 &= \mathbf{co\text{-}}\Sigma_1^0 = \mathbf{co\text{-}RE}. \end{aligned}$$

Other levels are defined inductively: for $i \geqslant 1$,

$$\begin{aligned} \Delta_{i+1}^0 &= \mathbf{R}^{\Sigma_i^0} = \{L: \text{ there is } B \in \Sigma_i^0 \text{ such that } L \text{ is recursive in } B\}, \\ \Sigma_{i+1}^0 &= \mathbf{RE}^{\Delta_i^0} = \mathbf{RE}^{\Sigma_i^0} = \{L: \text{ there is } B \in \Sigma_i^0 \text{ such that } L \text{ is r.e. in } B\}, \\ \Pi_{i+1}^0 &= \mathbf{co\text{-}}\Sigma_{i+1}^0. \end{aligned}$$

Sometimes they also define the 0-th level as $\Sigma_0^0 = \Pi_0^0 = \Delta_0^0 = \mathbf{R}$.

One can also define

$$\mathbf{AH} = \bigcup_{i \in \mathbb{N}} \Sigma_i^0 \ (= \bigcup_{i \in \mathbb{N}} \Pi_i^0 = \bigcup_{i \in \mathbb{N}} \Delta_i^0).$$

Exercise 5.1. Think why the two equivalences in the parentheses hold.

Note that the *class* denoted **AH** is not the same as the Arithmetical Hierarchy as a *set of classes*: the class **AH** is the union of all these classes.

5.3 An equivalent definition of AH using quantifiers

Theorem 5.1. *Let $i \geqslant 1$. Let $x, y_1, \ldots$ denote binary strings. Then*

1. *$L \in \Sigma_i^0$ if and only if there is $S \in \mathbf{R}$ such that for every x,*

$$x \in L \iff \exists y_1 \ \forall y_2 \ \ldots \ \mathbf{Q} y_i \ \langle x, y_1, \ldots, y_i \rangle \in S,$$

   where the quantifier $\mathbf{Q} = \exists$ if i is odd and $\mathbf{Q} = \forall$ otherwise.

2. *$L \in \Pi_i^0$ if and only if there is $S \in \mathbf{R}$ such that for every x,*

$$x \in L \iff \forall y_1 \ \exists y_2 \ \ldots \ \mathbf{Q} y_i \ \langle x, y_1, \ldots, y_i \rangle \in S,$$

   where the quantifier $\mathbf{Q} = \forall$ if i is odd and $\mathbf{Q} = \exists$ otherwise.

3. *$L \in \Delta_i^0$ if and only if $L \in \Sigma_i^0 \cap \Pi_i^0$.*

Proof. 1. Let us prove it by induction on i. The base of the induction is Theorem 1.1 and (1). Let us now prove the step ($i - 1 \mapsto i$).

$\Leftarrow$. If L is defined using quantifiers (using the same notation as in the statement of the theorem), let us construct an oracle DTM $M^\bullet$ and an oracle $O \in \Sigma_{i-1}^0$ that would together accept L.

On input x, the machine $M^\bullet$ attempts to find an appropriate y_1 by trying[6] all possible bit strings $y_1 \in \{0,1\}^*$

The process of checking a specific value of y_1, is simple: the machine wants to know whether

$$\forall y_2 \exists y_3 \ldots \langle x, y_1, y_2, \ldots \rangle \in S,$$

so it writes down the query $\langle x, y_1 \rangle$ and feeds it to the following oracle:

$$T = \{u : \exists y_2 \forall y_3 \ldots \langle x, y_1, y_2, \ldots \rangle \in \overline{S}\}$$

(the condition is just the negation of the question we are interested in).

By the induction hypothesis, $T \in \Sigma^0_{i-1}$ according to both definitions of Σ^0_{i-1} (note that since S is computed by a DTM that always stops, also $\overline{S} \in \mathbf{R}$). If the oracle answers "no", our machine $M^\bullet$ (running as M^T) stops and accepts. Otherwise it drops its attempt for this value of y_1.

Our machine (using the oracle T) accepts if and only if the appropriate value of y_1 exists, that is, it accepts the language L according to its quantifier-like definition in the statement of the theorem.

$\Rightarrow$. Now let L be defined using an oracle machine $M^\bullet$ and an oracle $O \in \Sigma^0_{i-1}$. By the induction hypothesis, there is $S' \in \mathbf{R}$ such that $\forall z \in \{0,1\}^*$

$$z \in O \iff \exists y_2 \forall y_3 \ldots \langle z, y_2, y_3, \ldots \rangle \in S'. \tag{5}$$

Let us write a program for deciding a new set $S \in \mathbf{R}$ that would allow us to give a quantifier-like definition for L.

To check that $x \in L$, we need to make sure that M^O accepts it. A witness given by the first existential quantifier will help us to verify it. Let us consider what do we need to include into this witness. The machine for S could run the machine $M^\bullet$ by itself given

- the time bound t (because this machine needs to stop while $M^\bullet$ may work forever),
- the oracle answers $a_1, a_2, \ldots, a_t \in \{0,1\}$ to M's queries (actually, fewer than that, but t answers definitely suffice, because $M^\bullet$ cannot make more queries within the time bound).

However, how this machine could verify that the oracle answers that someone supplied as a part of this witness are indeed correct? It is not an oracle machine! Thus let us include in the witness also a kind of "certificates" for the cases where $a_k = 1$ for the k-th query q_k:

- the value of y_2 such that $\forall y_3 \ldots \langle q_k, y_2, y_3, \ldots \rangle \in S'$ (cf. (5)).

[6] In the actual lecture I may have said "in parallel", but actually it is not needed, because the preprocessing step (writing $\langle x, y_1 \rangle$) takes a finite amount of steps and then the oracle answers immediately. Anyway, let me remind that we can run the processes for different values of y_1 "in parallel" (even they are not guaranteed to stop) so that in this "parallel" simulation each process that stops will come to its end in the simulation as well if the machine works long enough. (We have seen such simulations before: recall "zigzag" enumerations of pairs (t, z) for the number of steps t and the input z from the Automata course.)

Now, given x and a witness, we need to check (using $k-1$ quantifiers and the relation $S \in \mathbf{R}$ we are constructing) that

- $M^\bullet$ accepts x provided that we believe that the oracle answers are indeed $a_1, a_2, \ldots$ — this is easily done using a universal Turing machine.
- For every positive oracle answer $a_k = 1$ for a query q_k the certificate (the suggested value for y_2) is correct: we can check it using $k-2$ quantifiers starting with $\forall$ as

$$\forall y_3 \ldots \langle q_k, y_2, y_3, \ldots \rangle \in S' \tag{6}$$

- For every negative oracle answer $a_k = 0$ for a query q_k this answer is correct — we do not need a certificate for it, because this is done using $k-1$ quantifiers starting with $\forall$:

$$\forall y_2 \exists y_3 \ldots \langle q_k, y_2, y_3, \ldots \rangle \in \overline{S'}. \tag{7}$$

Overall, given x and the witness, we need to check the conjunction of many conditions and each of them is specified by a formula with at most $k-1$ quantifiers starting with $\forall$ and ending with a recursive predicate (language).

From the logic course, we know that if we have two quantified formulas: $\forall v_1 \exists v_2 \ldots A(\vec{u}, \vec{v})$ and $\forall v_1' \exists v_2' \ldots B(\vec{u}, \vec{v'})$ where the variables in $\vec{v}$ are distinct from those in $\vec{v'}$, then their conjunction is equivalent to the formula $\forall v_1 \forall v_1' \exists v_2 \exists v_2' \forall \ldots (A(\vec{u}, \vec{v}) \wedge B(\vec{u}, \vec{v'}))$. Thus let us "pack" formally all the quantified variables located at the same position in our (many) formulas with quantifiers

$$\begin{array}{r} \forall v_2^{(1)} \exists v_3^{(1)} \ldots \\ \forall v_2^{(2)} \exists v_3^{(2)} \ldots \\ \ldots \\ \forall v_2^{(i)} \exists v_3^{(i)} \ldots \\ \ldots \end{array}$$

as $V_2 = \langle v_2^{(1)}, v_2^{(2)}, \ldots, v_2^{(i)}, \ldots \rangle$, $V_3 = \langle v_3^{(1)}, v_3^{(2)}, \ldots, v_3^{(i)}, \ldots \rangle$, etc. Note that although we do not know how many strings we need to pack in a single V_j, it is not a problem, because we can encode all finite sequences (of varying but finite length) of bit strings (of varying but finite length) as bit strings (of varying but finite length), and our quantifiers, like $\forall V_2 \in \{0,1\}^*$, range over arbitrary long (albeit finite) strings.

In total, we can write the formula

$$\exists V_1 \forall V_2 \ldots \langle x, V_1, V_2, \ldots \rangle \in S$$

where V_1 is parsed as a time bound t, the oracle answers $a_1, \ldots, a_t$, and the witnesses $y_1^{(k)}$ for each (relevant) query q_k to be checked according to (6). The machine for S will use UTM for running $M^\bullet$ for t steps assuming that $a_1, \ldots, a_t$ are correct, and if this UTM accepts, then for every k from 1 to t it will check the correctness of a_k:

- if $a_k = 1$, it runs S' as in (6) using the input variables $v_\ell^{(k)}$ (extracted from V_ℓ) instead of y_ℓ in (6),
- if $a_k = 0$, it runs $\overline{S'}$ as in (7) using the input variables $v_\ell^{(k)}$ (extracted from V_ℓ) instead of y_ℓ in (7).

If all these checks are successful, then S accepts; otherwise it rejects.

2. To show the equivalence for Π_i^0 as a corollary of the equivalence for Σ_i^0, use the fact that $\Pi_i^0 = \mathbf{co}\text{-}\Sigma_i^0$ and that negating a quantified formula followed by $\langle \ldots \rangle \in S$ switches the quantifiers and transforms the decidable language S into the language $\overline{S}$, which is also decidable.

3. The last item $\Delta_i^0 = \Sigma_i^0 \cap \Pi_i^0$ is analogous to $\mathbf{R} = \mathbf{RE} \cap \mathbf{co}\text{-}\mathbf{RE}$, yet with oracles. It will be proven in a problem session. □

5.4 Examples of languages in AH

Example 5.1.

$$\begin{aligned} L_{FIN} &= \{\langle M \rangle : |L(M)| < \infty\} = \{\langle M \rangle : \exists n \in \mathbb{N}\ \forall x \in \{0,1\}^*\ (|x| > n \Rightarrow x \notin L(M)\} \\ &= \{\langle M \rangle : \exists n \in \mathbb{N}\ \forall x \in \{0,1\}^*\ \forall t \in \mathbb{N}\ (|x| \leqslant n \vee M \text{ does not accept } x \text{ in } t \text{ steps}\} \in \Sigma_2^0. \end{aligned}$$

Note that two similar quantifiers like $\forall x \in \{0,1\}^*\ \forall y \in \{0,1\}^*$ can always be merged into a single quantifier $\forall z \in \{0,1\}^*$, where $z = \langle x, y \rangle$ is an appropriate encoding of pairs.

More examples will be added when/if they are given in the other group.

5.5 Complete problems

Recall that by default completeness of languages in computability classes is considered under many-one reductions. For example, a problem is called Σ_k^0-complete if it is complete for Σ_k^0 *under many-one reductions*.

Let us construct complete languages for Σ_k^0 (for every $k \geqslant 1$) inductively. Consider

$$\begin{aligned} C_1 &= \{\langle M, x \rangle\ : M(x) = 1\}, \\ C_k &= \{\langle M^\bullet, x \rangle : M^{C_{k-1}}(x) = 1\} \quad \text{(for every } k \geqslant 2). \end{aligned}$$

Thus C_1 is just `AP`.

Theorem 5.2. *For every $k \geqslant 1$, C_k is Σ_k^0-complete.*

Proof. We need to prove that $C_k \in \Sigma_k^0$ and also that it is Σ_k^0-hard. We will now prove it by induction on k.

Base: $k = 1$. As we learned earlier, $C_1 =$ `AP` is complete for $\Sigma_1^0 = \mathbf{RE}$ (Theorem 3.1).

Step: $k-1 \mapsto k$.

1. To show that $C_k \in \Sigma_k^0$, generalize the notion of a UTM so that it could run oracle machines (and access the [same] oracle itself). That is, $U^\bullet(M^\bullet, x)$ simulates $M^\bullet$ like an ordinary UTM, but when it has to execute the instruction of accessing the oracle (that is, $M^\bullet$ comes to the state q_{ask}), it asks the oracle itself and updates the configuration of the simulated machine according to the oracle's answer.

Then $C_k = \{\langle M^\bullet, x\rangle : M^{C_{k-1}}(x) = 1\} = \{\langle M^\bullet, x\rangle : U^{C_{k-1}}(M^\bullet, x) = 1\}$, thus it is indeed accepted by an oracle DTM (namely, $U^\bullet$) given access to an oracle in C_{k-1}.

2. We will now prove that C_k is Σ_k^0-hard. Consider a language $L \in \Sigma_k^0$. Let us construct a many-one reduction from L to C_k.

By the definition of Σ_k^0, there is an oracle DTM $M^\bullet$ and there is an oracle $O \in \Sigma_{k-1}^0$ such that $L(M^O) = L$. By the induction assumption, $O \leqslant C_{k-1}$, that is, there is a many-one reduction f computable by some DTM F (that always stops), and by definition $\forall y\ y \in O \iff f(y) \in C_{k-1} \iff F(y) \in C_{k-1}$.

Let us consider the following oracle machine $T^\bullet$ with input x:

- Run as $M^\bullet$, but instead of every oracle query q do the following:
  - Compute $z := f(q)$ using F.
  - Query the oracle about z.

That is, $T^\bullet$ "preprocesses" oracle queries using the reduction, because it intends to query C_{k-1} instead of O.

Here is a reduction from L to C_k:

$$r(x) = \langle T^\bullet, x\rangle.$$

Indeed, it is easily computable by a DTM that always stops, and, for every x,

$$x \in L \iff M^O(x) = 1 \iff T^{C_{k-1}}(x) = 1 \iff \langle T^\bullet, x\rangle \in C_k \iff r(x) \in C_k.$$

The second equivalence holds because the preprocessing step ensures that $T^{C_{k-1}}(x) = M^O(x)$. □

Corollary 5.1. *For every $k \geqslant 1$, the language $\overline{C_k}$ is Π_k^0-complete.*

Proof. Recall that $\Pi_k^0 = \mathbf{co}\text{-}\Sigma_k^0$, thus $\overline{C_k} \in \Pi_k^0$.

Consider a language $L' \in \Pi_k^0$. Its complement $\overline{L'}$ is in Σ_k^0 and thus it is many-one reducible to C_k. Recall that if a many-one reduction reduces a language A to a language B, then it also reduces $\overline{A}$ to $\overline{B}$. Thus L' is reducible to $\overline{C_k}$. □

Unlike the classes Σ_k^0 and Π_k^0, **AH** has no complete problems. To show that, let us make two observations:

Lemma 5.1. *Every class Σ_k^0 (as well as Π_k^0 and Δ_k^0) is closed under many-one reductions.*

Proof. If $L \leqslant L'$, the machine for L runs the reduction in order to "preprocess" the input and then works as the machine for L'. By the definition of a many-one reduction (in particular, given that the reduction always stops), this machine solves L correctly and has the same stopping / non-stopping behaviour as the machine for L' (that is, for Σ_k^0 is accepts every input in the language and does not accept anything else, and for Δ_k^0 it also always stops). □

Lemma 5.2. *For every $k \geqslant 1$, $\Sigma_k^0 \neq \Sigma_{k+1}^0$ and $\Sigma_k^0 \neq \Pi_k^0$.*

Proof. By diagonalization analogous to **RE** $\neq$ **R** (thus **RE** $\neq$ **co-RE**), but now with an oracle. It will be proved in practical sessions. □

Theorem 5.3. **AH** *has no complete languages.*

Proof. Assume that there is a language C complete for **AH**. This class is the union of classes Σ_k^0, so there is some specific k such that $C \in \Sigma_k^0$.

Consider any $L \in \Sigma_{k+1} \setminus \Sigma_k$ (existing by Lemma 5.2). Since C is complete (and thus hard) for **AH**, this language L is many-one reducible to C. By Lemma 5.1 it means that also $L \in \Sigma_k^0$, a contradiction! □

Remark 5.2. In this proof we are effectively showing that if **AH** possesses a complete language, then starting from a certain value of k

$$\Sigma_k^0 = \Sigma_{k+1}^0 = \Sigma_{k+2}^0 = \dots .$$

This phenomenon is called a collapse of this hierarchy. While for the Arithmetical Hierarchy we have shown that it simply cannot happen, for some other hierarchies we do not know it, and the absence of a *collapse* is a frequently used assumption.

6 RE-intermediate languages

We know that in **RE** there are "easy" languages (**R**) and "hard" languages (**RE**-complete). Is there anything in between? The following theorem is due to Emil Post:

Theorem 6.1 (Post, 1944)**.** *There exists a non-***RE***-complete language in* **RE** $\setminus$ **R**.

We will call such languages **RE**-intermediate. In this section we will prove Post's theorem. A similar theorem (due to Ladner) for **NP**-intermediate languages is, of course, conditioned on **P** $\neq$ **NP** (otherwise all nontrivial languages in **NP** are **NP**-complete), and is beyond the scope of this course.

6.1 Immune and simple sets

The construction of the desired language is based on the concept of an immune set which is even more important that the theorem itself. This concept can be used with respect to many computability and complexity classes. We need it with respect to **RE**:

Definition 6.1 (**RE**-immune sets)**.** An infinite language L is called **RE**-immune if it contains no infinite **RE** subsets, i.e., $\forall L' \subseteq L$ ($|L'| = \infty \Rightarrow L' \notin \mathbf{RE}$).

An **RE**-intermediate language we construct is a "simple set", which is an **RE** set with an **RE**-immune complement.

Definition 6.2 (Simple sets)**.** A language S is simple if $S \in \mathbf{RE}$ and $\overline{S}$ is **RE**-immune.

We first construct a simple set and then show that simple sets are **RE**-intermediate.

Lemma 6.1 (Lemma 3 in the videos)**.** *There exists a simple set.*

Proof. Recall that a language belongs to **RE** if and only if it has an enumerator: a DTM that prints all words of this language (and only them) one by one (possible printing one word several times).

Such DTMs can be given Turing numbers (corresponding to their encodings), so let us call them $E_1, E_2, \ldots$.

Our language S will be defined by its enumerator. Let us described a program for it:

> We intend to run all the enumerators "in parallel", that is, run infinitely many processes corresponding to all possible Turing numbers (that is, natural numbers) in a controlled way such that for each i and j sooner or later we will simulate the step j of the process i. (We did it many times already.)
>
> **When running E_i, let us look at what it prints. Once it prints the first (for it) number $j \geqslant 2i$, stop this process (other processes continue running!) and print j.**
>
> For each i, let us denote this number $\boldsymbol{j_i}$ (for some i it may be undefined, because some enumerators correspond to finite languages) — we will use this notation in the following proof.

Let us prove that S is simple. First of all, $S \in \mathbf{RE}$, because it is defined by an enumerator. Its complement is infinite: for each $n \in \mathbb{N}$, the interval $\{1, 2, \ldots, 2n - 1\}$ contains at most $n - 1$ numbers j_i (namely, it may contain $j_1, \ldots, j_{n-1}$), because for $k \geqslant n$ by definition $j_k \leqslant 2k \leqslant 2n$.

Now assume to the contrary that $\overline{S}$ is not **RE**-immune. Then it has an infinite recursively enumerable subset $X \subseteq \overline{S}$. But X also has some enumerator E_x, and it was run by our enumerator in the corresponding process. Since $|X| = \infty$, sooner or later E_x had to print its first number y that is at least $2x$, and thus our enumerator for S would print this y as well.

It follows that $y \in S$, but E_x prints X, therefore $y \in X \subseteq \overline{S}$ — a contradiction! □

It is easy to see that a simple set cannot belong to **R**:

Lemma 6.2 (Lemma 2 in the videos)**.** *Simple sets are not recursive.*

Proof. If S is simple, then $\overline{S}$ is **RE**-immune, which means also that $\overline{S} \notin \mathbf{RE}$. Thus, in particular, $\overline{S} \notin \mathbf{R}$, and $S \notin \mathbf{R}$. □

Demonstrating that a simple set cannot be **RE**-complete is a bit harder and requires a new notation introduced in the next section.

6.2 Productive sets

Let us consider all DTMs $M_1, M_2, \ldots$ that compute functions (that is, may produce more than just a single bit of output) but not necessarily stop (in which case the function value is undefined). They can also[7] be given Turing numbers, of course (corresponding to their descriptions).

We identify natural numbers with bit strings (say, by identifying a bit string $b_1 \ldots b_k$ with the Turing number with binary representation $1b_1 \ldots b_k$).

For $x \in \mathbb{N}$, define

$$W_x = \{y : M_x(y) \neq \nearrow\},$$

that is, W_x is the set of strings where the function computed by M_x (a DTM with Turing number x) is defined. (If M_x stops on every string, then $W_x = \{0,1\}^*$, and this function is defined everywhere, that is it is total[8])

Definition 6.3 (Productive sets)**.** A language L is productive with a productive function $\sigma : \{0,1\}^* \to \{0,1\}^*$ if for every x <u>such that $W_x \in L$</u>, this function maps x to L avoiding W_x, that is,

$$\sigma_x(x) \in L \setminus W_x.$$

Let us give an important example of a productive set. Let $\mathcal{K} = \{x : x \in W_x\}$, that is, a Turing number belongs to this set if the corresponding machine stops on its own Turing number.

Lemma 6.3 (Lemma 1 in the videos)**.** *$\overline{\mathcal{K}}$ is productive with the identity function $\sigma(x) = x$ as its productive function.*

Proof. Assume to the contrary that it is not the case, that is, there exists x such that $W_x \subseteq \overline{K}$, but $\sigma(x) \notin \overline{K} \subseteq W_x$.

Consider the two possible cases:

1. If $x \in W_x$, then $x \in \mathcal{K}$ by the definition of $\mathcal{K}$, which contradicts our assumption $W_x \subseteq \overline{K}$.

2. If $x \notin W_x$, then $x \in \overline{\mathcal{K}}$ by the definition of $\mathcal{K}$. However, $\sigma(x) = x$, so it contradicts our assumption $\sigma(x) \notin \overline{K} \subseteq W_x$.

[7] In fact, a Turing number is given to a Turing machine without any condition on what it computes, what it outputs, whether it stops or not. Therefore, Turing numbers given to enumerators, to DTMs computing functions, and to DTM deciding languages are the very same Turing numbers.

[8] Normal mathematicians — not logicians — would simply call it a function, as opposed to a partial function that may be undefined for some values of its arguments.

□

It remains two prove two more lemmas:

Lemma 6.4 (announced as Lemma 4 in the videos)**.** *If S is simple, then $\overline{S}$ is not productive.*

Lemma 6.5 (announced as Lemma 5 in the videos)**.** *If C is* **RE***-complete, then $\overline{C}$ is productive.*

These two lemmas complete our proof of Theorem 6.1:

Proof of Theorem 6.1. We proved that simple sets exist (Lemma 6.1) and they do not belong to **R** (Lemma 6.2). They are not productive by Lemma 6.4, and all non-productive sets are non-**RE**-complete by Lemma 6.5. □

Proof of Lemma 6.4 (Lemma 4 in the videos). Assume that $\overline{S}$ is productive with a productive function σ and let us prove that S is not simple. Namely, we will prove that $\overline{S}$ is not **RE**-immune: We will construct an infinite set $B \subseteq \overline{S}$ such that $B \in \mathbf{RE}$.

We will construct B using an enumerator, so it will be obvious that $B \in \mathbf{RE}$. Consider the following sequence of sets and machines:

(0) Let i_0 be the Turing number of some machine M_{i_0} that never stops, thus $W_{i_0} = \emptyset$.

(1) Since $W_{i_0} \subseteq \overline{S}$, the function σ will give us an element of $\overline{S} \setminus W_{i_0}$.
Let us print $\sigma(i_0)$ and construct a new machine M_{i_1} with $W_{i_1} = W_{i_0} \cup \{\sigma(i_0)\}$. The pseudocode for this machine is:

On input x, if $x = \sigma(i_0)$, then accept, otherwise work like M_{i_0}.

Note that $W_{i_1} \subseteq \overline{S}$.
. . .

$(k+1)$ Since $W_{i_k} \subseteq \overline{S}$, the function σ will give us an element of $\overline{S} \setminus W_{i_k}$.
Let us print $\sigma(i_k)$ and construct a new machine $M_{i_{k+1}}$ with $W_{i_{k+1}} = W_{i_k} \cup \{\sigma(i_k)\}$:
The pseudocode for this machine is:

On input x, if $x = \sigma(i_0)$, then accept, otherwise work like M_{i_k}.

Note that $W_{i_{k+1}} \subseteq \overline{S}$.

By construction every time σ gives us a new element of $\overline{S}$, so our enumerator will print an infinite recursively enumerable subset of $\overline{S}$, thus $\overline{S}$ is not **RE**-immune and S is not simple. □

Proof of Lemma 6.5 (Lemma 5 in the videos). Assume that C is **RE**-complete, let us build a productive function σ for $\overline{C}$.

Recall set $\mathcal{K} = \{x : x \in W_x\}$ from Lemma 6.3. Since $\mathcal{K} \in \mathbf{RE}$ (to check $x \in \mathcal{K}$ one needs to run $M_x(x)$ and accept if it stops), there is a many-one reduction f from $\mathcal{K}$ to C.

Let us consider some input i and let us decide (and compute!) where our productive function σ will map it. Note that we are mostly interested in the case where $W_i \subseteq \overline{C}$ (because otherwise there is no condition on the behaviour of σ). Consider $f^{-1}(W_i) := \{x : f(x) \in W_i\} \subseteq \overline{\mathcal{K}}$ (the containment is by the definition of many-one reductions). We will characterize it in a different way.

Consider the following machine T_i:

```
On input x,
-- compute y := f(x),
-- simulate (using UTM) M_i(y) and return the same answer.
```

Let us define a (total, computable) function τ that on input i returns the[9] Turing number of T_i. (That is, τ transforms machines into machines.)

Note that for every x,

$$x \in W_{\tau(i)} \iff T_i \text{ stops on } x \iff M_i \text{ stops on } f(x) \iff f(x) \in W_i \iff x \in f^{-1}(W_i),$$

that is: $W_{\tau(i)} = f^{-1}(W_i)$.

Let us define $\sigma(i) := f(\tau(i))$ and show that it is productive for $\overline{C}$.

Assume that $W_i \subseteq \overline{C}$.

First we need to show that $\underline{\sigma(i) \in \overline{C}}$. Since $W_i \subseteq \overline{C}$, we have $f^{-1}(W_i) \subseteq \overline{\mathcal{K}}$, i.e., $W_{\tau(i)} \subseteq \overline{\mathcal{K}}$. Since $id(x) = x$ is a productive function for $\overline{\mathcal{K}}$ by Lemma 6.3, $\tau(i) \in \overline{\mathcal{K}} \setminus W_{\tau(i)}$, thus $\sigma(i) = f(\tau(i)) \in f(\overline{\mathcal{K}} \setminus W_{\tau(i)}) \subseteq \overline{C}$.

Now let us show that $\underline{\sigma(i) \notin W_i}$. By the above, $\sigma(i) \in f(\overline{\mathcal{K}} \setminus W_{\tau(i)}) = f(\overline{\mathcal{K}} \setminus f^{-1}(W_i))$, i.e., $\exists x \in \overline{\mathcal{K}}\ (\sigma(i) = \underline{f(x)} \wedge \forall y \in W_i\ \underline{f(x)} \neq y)$, in particular, $\sigma(i)$ is different from every $y \in W_i$. $\square$

7 Gödel's incompleteness theorem — a computability point of view

The material of this section is optional and will not be included into the exam.
You can consider it as an a yet another example of a **RE** set:
a set of statements provable in a proof (derivation) system.

What is a mathematical proof? This is something that another mathematician can verify without having to perform creative work. In other words, this is something that could be checked algorithmically. A particular case of this concept is an axiomatic proof system that consists of axioms and derivation rules; the use of all of them is algorithmically checkable. Such a proof is a sequence of lines (closed logical formulas), where the next line is either an axiom or the result of an application of a derivation rule to earlier derived lines.

[9] We assumed that every machine has infinitely many Turing numbers, because we can add garbage to the description. Here we mean some particular Turing number, i.e., without any garbage — simply the number corresponding to the encoding of the most standard implementation of the pseudocode above.

Example 7.1 (Peano arithmetic)**.** As an example, look at a first-order formulation of Peano arithmetic, a system intended to formalize natural numbers. It has an equivalence relation = (we will omit its axiom, they are assumed), binary functions + and $\cdot$, a constant 0, and an unary function S intended to produce the next natural number (that is, to write i one writes $\underbrace{S(\ldots(S}_{i \text{ times } S(\ldots)}(0))\ldots))$:

(This is just an example! You do not need it for the exam.)

$$\begin{array}{ll}
\forall x & (0 \neq S(x)) \\
\forall x \forall y & (S(x) = S(y) \Rightarrow x = y) \\
\forall x \forall y & (x + S(y) = S(x + y)) \\
\forall x \forall y & (x \cdot S(y) = x \cdot y + x) \\
\forall \bar{y} & \Big(\Big(\varphi(0, \bar{y}) \wedge \forall x \big(\varphi(x, \bar{y}) \Rightarrow \varphi(S(x), \bar{y}) \big) \Big) \Rightarrow \forall x \varphi(x, \bar{y}) \Big)
\end{array}$$

Here $\bar{y}$ stands for any finite sequence of variables; φ may be any Boolean formula in this language, for example[a], $\exists y \forall z\ (y = z \vee S(S(0)) = S(z \cdot y))$.

The derivation rules are *modus ponens*:

$$\frac{F,\ F \Rightarrow G}{G},$$

the rule allows to substitute the outermost universal quantifier by a term[b] t:

$$\frac{\forall x\ F}{F|_{x=t}}$$

(that is, replacing all occurrences of x by (t)), and other logical rules for $\exists$, $\wedge$, $\vee$, $\Rightarrow$, etc, that you can find in any textbook on mathematical logic.

[a] This example of a formula has no particular sense.

[b] A term is a constant, a variable, or the result of an application of a function to terms.

Since mathematical proofs in a particular proof system can be algorithmically checkable, it makes the set of theorems (provable, derivable facts) *recursively enumerable*.

One can formalize the notion of a proof as follows (think of L as a language consisting of [provable] theorems):

Definition 7.1 (Proof system)**.** Let L be a language. An algorithm (a DTM that stops on every input) V defines a proof system for L if the following two conditions hold for every string $x \in \{0,1\}^*$:

(Soundness) For every $w \in \{0,1\}^*$, if $V(x, w) = 1$, then $x \in L$.
(If w accepts a proof, then x is a proven theorem.)

(Completeness) If $x \in L$, then $\exists w \in \{0,1\}^*\ V(x, w) = 1$.
(If x is a theorem, there is w that V accepts as a proof of x.)

We are interested in proof systems for some language L consisting of (closed) Boolean formulas, and such that for every (closed) Boolean formula F either $F \in L$ or $\neg F \in L$. Gödel's incompleteness theorem says that not only the axioms and derivation rules listed in Example 7.1 constitute an incomplete proof system, but there is no sound and complete proof system for the language of integer arithmetic (that is, for the set of strings that encode all closed Boolean formulas valid for natural numbers, as we know them, that is, satisfying at least the axioms and derivation rules listed in the example) such that all its axioms and rules are algorithmically checkable (and even recursively enumerable).

In the epoch of Gödel, it was a nontrivial and a very surprising (and disappointing) fact suggesting that even if we agree on some properties of natural numbers expressible like above, we will never arrive at a proof system that would either prove or disprove every logical sentence about them. After the notions of an algorithm, **R**, **RE**, and even **AH** have been formalized, the proof of this theorem became rather technical. Here is an informal exposition of a strategy that allows to prove it:

As we noticed, any language possessing a sound and complete proof system is recursive enumerable (that is, characterizable via a decidable language S as $\exists w\, S(\langle x, w\rangle)$ with a single quantifier $\exists w$). On the other hand, arithmetic formulas can have any number of quantifiers followed by a formula about natural numbers. In a practical session it was proved that $\Sigma_i^0 \neq \Sigma_{i+1}^0$ for every i. From Theorems 5.1 and 5.2 we know that a complete problem for Σ_i^0 is given by i quantifiers followed by a decidable predicate. Therefore, it remains to show that arithmetic formulas can "simulate" all DTMs, i.e., decidable predicates (then one can add any number of quantifiers that would definitely put the task of checking such a statement outside **RE**). We will not prove it formally; just note that the result of an execution of a Turing machine M on input x can be expressed as

$$\exists n \in \mathbb{N}\ \exists \vec{C}\ Start(C_0) \wedge Accept(C_{|\vec{C}|}) \wedge \forall i \leqslant |\vec{C}|\ Step(C_{i-1}, C_i),$$

where $\vec{C}$ is a vector of configurations of M, $|\vec{C}|$ is the number of configurations in it, $Start$ is the predicate checking that the configuration is the starting one and contains x as the input; $Accept$ is the predicate checking that the configuration is the accepting one (the machine state is q_Y); $Step$ is the predicate checking that the two given configurations are a correct step of the execution of M. It remains to express these predicates as logical formulas about natural numbers; this is doable but somewhat technical, so we will not do it in this course.

8 Introduction to Kolmogorov complexity

We know strings look different. Even though every string of length n (even 0^n) has the same chance to be produced by a uniform random generator, some of the strings look "random" to us (without any "law" or "order" in them) and some look structured — that is, we can describe them we fewer than n bits.

One can think about different methods of *compressing* strings (literally, of *uncompressing* a string from its description), such as Lempel-Ziv encoding or anything else. One common

thing about these methods is that they are *algorithms*. It gives rise to the following definition of Kolmogorov complexity:

Definition 8.1 (Kolmogorov complexity — informal)**.** The Kolmogorov complexity of a string x is the length of the shortest program printing this string x.

Of course, different "hardware" (DTM, RAM, C++ program) will produce different Kolmogorov complexity values. However, if we stick to models of computations that allow us to store constants (constant strings) without any overhead (a bit string of size n is stored using n bits and not, say, $2n$ bits), it turns out that they will produce essentially the same number — up to an additive constant ($+O(1)$). It will be discussed in more depth in the tutorial (practical session).

Because of that it is reasonable to discuss the Kolmogorov complexity of a string up to an additive constant. Note that we are interested in the Kolmogorov complexity of individual strings, not languages! Even if we are talking about asymptotic bounds, these are bounds on the Kolmogorov complexity of (a series of) individual strings.

Let us stick to a specific definition of Kolmogorov complexity (the same as in Sipser's book [Sip]):

Definition 8.2 (Kolmogorov (descriptional) complexity — formal)**.** The Kolmogorov complexity of a string $x \in \{0,1\}^*$ is defined as

$$K(x) := \min\{|\langle M, w\rangle| : M(w) = x\}.$$

Thus our model of computation is a "self-uncompressing archive": it includes a "compressed string" w and the "uncompressing machine" M.

Despite one can save the information here both in M and in w, think about the first part (M) as the short one and the second part (w) as the main one. In any case, the encoding of the pair $\langle M, w\rangle$ can be done using $2|\langle M\rangle| + 2 + |w|$ bits by repeating each bit of M twice and writing "the stop sequence" 01 after that separating the encoding of M from w. In the beginning of the course (Section 1.1) we talked about more efficient encodings of pairs, but what is essential for us now is that the overhead is proportional to the length of $\langle M\rangle$ (which is usually short).

Let us observe a natural fact: there exist incompressible strings of any length:

Proposition 8.1. *$\forall n \in \mathbb{N}\ \exists x \in \{0,1\}^n\ K(x) \geqslant n$.*

Proof. Each description corresponds to a single string only. There are $1+2+4+\ldots+2^{n-1} = 2^n - 1$ descriptions of length at most $n-1$. Since there are 2^n strings of length n, at least one of those does not have any description of length at most $n-1$. □

Now what about *compressible* strings? In the next example we look at very easy strings.

Example 8.1. What is $K(1^n)$? It can certainly depend on n, but in any case $K(1^n) = \log_2 n + O(1)$. To see it consider the following program:

```
Program B:
On input w (in binary),
-- for i := 1 to w print 1.
```

Since $\langle B, n\rangle$ indeed describes 1^n (namely, $B(n) = 1^n$, it indeed prints the desired string), we have that

$$K(1^n) \leqslant 2|B| + 1|\lceil \log_2 n\rceil$$

(recall the discussion above on the encoding of pairs; you can use a better bound on the length of such encoding), that is,

$$K(1^n) = \log_2 n + O(1).$$

Note that it is only an upper bound on $K(1^n)$. For some values of n it can be much less, for example, observe that if $N = 2^{2^n}$, then $K(1^N) = \log_2 n + O(1)$ despite of the fact that this string is so much longer than 1^n.

We frequently relate the Kolmogorov complexity of one string to the Kolmogorov complexity of another string. As an example let us consider strings of the form xx, that is, the concatenation of two copies of x, and let us estimate the Kolmogorov complexity of such a string in terms of the Kolmogorov complexity of the original string x.

Proposition 8.2. $\forall x \in \{0,1\}^*\ K(xx) = K(x) + O(1)$.

Proof. Let $\langle M, w\rangle$ be the minimal description of x, that is, $K(x) = |\langle M, w\rangle|$. Let us write a program that would use x's description (even not mimimal) $\langle M, w\rangle$ in order to print xx.

On input $\langle T, u\rangle$, our program D does the following:

```
Program D.
On input ⟨T, u⟩,
-- Simulate T(u) using UTM, it will give us a string z,
-- Output zz.
```

Now $\langle D, \langle M, w\rangle\rangle$ is a valid description of xx: it will run $M(w)$, get x, and write down two copies of it. The length of this description is $O(|\langle D\rangle|) + |\langle M, w\rangle|$. Since D is a specific machine (not depending on M or w), its program has a constant size, thus we have just proved that $K(xx) = K(x) + O(1)$. ☐

Let us generalize this example in the next exercise:

Exercise 8.1 (to be partially solved in practical sessions). Prove that

1. $\forall x, y \in \{0,1\}^*\ K(xy) = K(x) + K(y) + O(\log K(x))$.

2. $\forall c \in \mathbb{N}\ \exists x, y \in \{0,1\}^*\ K(xy) > K(x) + K(y) + c$,
   i.e., the bound in the previous item cannot be improved to a constant.

Exercise 8.2 (to be solved in practical sessions)**.** There is no DTM that would compute $K(x)$ from x.

9 Search to decision reductions

9.1 Search problems

Recall Def. 1.2: A language L belongs to the class **NP** if there is $S \in \mathbf{P}$ and a polynomial p such that for every $x \in \{0,1\}^*$,

$$x \in L \iff \exists w \in \{0,1\}^{\leqslant p(n)}\ \langle x, w\rangle \in S$$

Note that S can be viewed both as a language (of pairs $\langle x, w\rangle$) and as a binary relation (between x's and w's). For a given input instance x, the question $x \in^? L$ is a decision problem. One can ask for more: instead of just outputting the answer "yes", output also a solution (or witness) w that satisfies $\langle x, w\rangle \in S$. This is called a search problem associated with the relation S (or just "search problem S"). One can define a search problem for any relation $S \subseteq \{0,1\}^* \times \{0,1\}^*$. If the relation $S \in \{0,1\}^n \times \{0,1\}^{\leqslant}\ p(n)$ is polynomial-time computable ($S \in \mathbf{P}$ as a language), then we say that this search problem is an **NP**-search problem.

Definition 9.1. A search problem associated with a binary relation S asks on input x to output w such that $S(x, w) = 1$.

We call it an **NP**-search problem if its solutions are polynomially bounded in length ($S \in \{0,1\}^n \times \{0,1\}^{\leqslant}\ p(n)$ for a polynomial p) and can be verified in polynomial time ($S \in \mathbf{P}$).

The class of **NP**-search problems is denoted **Search-NP** or **FNP**.

Some sources require also to say "No solution" explicitly if there is no solution. Note that for an **NP**-search problem it does not matter because we can verify solutions efficiently.

For each search problem S one can define the corresponding language of positive instances

$$L_S = \{x : \exists w\ \langle x, w\rangle \in S\}.$$

Remark 9.1. Note that while L_S is defined by S unambiguously, there are many relations that define the same language (i.e., there are relations $S' \neq S$ such that $L_{S'} = L_S$). Moreover, they may have very different hardness: the relation

$$\mathtt{FACTOR}(x, w) = (1 < w < x \wedge x \equiv 0 \pmod{w})$$

asks to find a nontrivial divisor w of x (we identify bit strings x, w with the corresponding natural numbers). This search problem is assumed to be very hard (note that the length

of the input here is $\log_2 x$): if one succeeds in solving it, the widely known cryptosystem RSA would be broken. On the other hand, the corresponding language $L_{\texttt{FACTOR}}$ consists of all composite numbers, and it is known to be decidable in polynomial time (this is a nontrivial result by Agrawal, Kayal, and Saxena (2002)). Therefore, the relation

$$\texttt{TRIVCOMP}(x, z) = (\exists w \ \ x \equiv 0 \pmod{w})$$

(note that z has nothing to do with w!) is also an **NP**-search problem, and it can be solved in polynomial time. Note that $L_{\texttt{FACTOR}} = L_{\texttt{TRIVCOMP}} = \{$ composite numbers $\}$, thus the hardness of the search problem FACTOR does not stem from the hardness of the corresponding language — the language is easy! It is just an intricate search problem formulating this easy language in computationally difficult terms.

Definition 9.2. A search-**NP** problem belongs to the class **Search-P** (also denoted $\boldsymbol{f}\mathbf{P}$, but this abbreviation sometimes denotes a different class, so we won't use it) if there is a polynomial-time DTM that solves it.

Note that we consider it for **Search-NP** problems only, not for general search problems.

9.2 A search-to-decision reduction for Circuit-SAT

What if we have an efficient algorithm for deciding a language? Can it help us to solve the search problem for a corresponding relation? While it is not the case for the factoring problem, it turns out that for **NP**-complete languages we can reduce search to decision. We will start with the corresponding very general definition and with a particular language of Circuit-SAT.

Recall Def. 3.6 where we defined Turing (oracle) reductions for languages. One can define them for any other kind of problem as well. Let us do it for the polynomial-time version:

Definition 9.3 (Turing reductions)**.** We say that a problem W is polynomial-time Turing-reducible to a problem S (denoted $W \leqslant_T^p S$) if there is an oracle DTM $R^\bullet$ that stops in a polynomial number of steps[10] and such that if an oracle O correctly solves S, then R^O correctly solves W.

Note that in this definition the oracle may be a language but may be also a black box that output more than a yes/no answer, that is, it can write more than a single bit on the oracle tape of R^O when asked.

This way we can reduce a search problem to decision problem (or another search problem, etc).

Theorem 9.1. *Consider* $R_{\texttt{CircuitSAT}} = \{\langle C, w\rangle : C(w) = 1\}$. *The search problem for it polynomial-time Turing-reduces to the decision problem for* Circuit-SAT $= L_{R_{\texttt{CircuitSAT}}}$.

[10] For every input and for every possible oracle.

Proof. Assume for simplicity that our circuits may have constant gates (that is, the two 0-ary functions always outputting a constant (`False` or `True`)). Then given a circuit D and its input x_i, we can construct a new circuit $D|_{x_i=1}$ by replacing its input x_i with the constant gate "`True`" (and similarly for $D|_{x_i=0}$).

The idea of our algorithm (oracle DTM) is very simple: we will ask yes/no queries to the oracle for `Circuit-SAT`, and the answers will guide us through a single path in the brute-force search tree, resulting in a satisfying assignment.

Here is a pseudocode for $R^\bullet$ (it uses recursion):

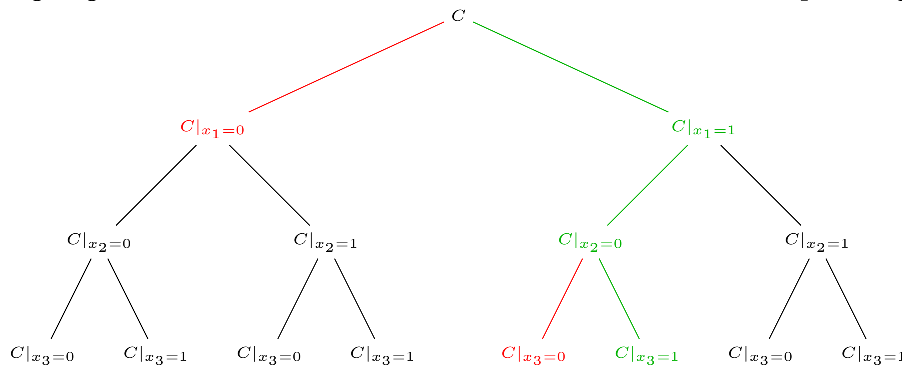
```
Algorithm Search using oracle Decision.
Input:  circuit C.

If Decision(C)=0, then return ‘‘No solution’’.     // To avoid searching for nothing

If C has no inputs, stop.                                      // End of recursion

Let x_1 be the first variable^11 in C.
If Decision(C|_{x_1=0})=1
-- Then print ‘‘x_1 = 0’’ and run Search(C|_{x_1=0})       // Dive into the left subtree
-- Else print ‘‘x_1 = 1’’ and run Search(C|_{x_1=1}).     // Dive into the right subtree
```

In the following illustration this algorithm follows the "green" path only because of receiving negative answers from the oracle on "red" circuits corresponding to unsatisfiable subtrees.



Note that this procedure will stop after making at most $2n+1$ queries, where n is the number of variables (input bits), and its running time is polynomial in the length of $\langle C\rangle$. □

9.3 Search-to-decision reductions for other languages

Theorem 9.2. *Consider an* **NP**-*search problem associated with a relation S corresponding to an* **NP**-*complete language $L_S = \{x : \exists w \;\; S(x,w)=1\}$. Then the search problem for S is polynomial-time Turing-reducible to the decision problem for L_S.*

[11] As we go inside the recursion, x_1 will correspond to different variables! Our next circuits will have fewer and fewer of them.

To prove this theorem, we combine three polynomial-time reductions into one:

$$S \quad \longrightarrow^p_{???} \quad R_{\texttt{Circuit-SAT}} \quad \longrightarrow^p_T \quad \text{(decision) } \texttt{Circuit-SAT} \quad \longrightarrow^p_m \quad L_S$$

The second reduction is the search-to-decision reduction for `Circuit-SAT` due to Theorem 9.1.

The third reduction is due to the **NP**-completeness of L_S.

It remains to show the first reduction, that is, to reduce the search problem for an **NP**-search problem S to the search problem for `Circuit-SAT`, that is, to demonstrate that `Circuit-SAT` is not just an **NP**-hard decision problem, but also demonstrates its hardness properties as a search problem. We will see it in the next section while proving the Cook–Levin Theorem, namely, the **NP**-hardness of `Circuit-SAT` (Lemma 10.2 and Remark 10.3).

10 The Cook-Levin Theorem: A proof

Theorem 10.1 (S.A.Cook, L.A.Levin)**.** `3-SAT` *is* **NP***-complete.*

We have already made one stop towards the proof of this theorem: In Lemma 3.3 we constructed a polynomial-time many-one reduction of `Circuit-SAT` to `3-SAT`. It remains to prove that `Circuit-SAT` is **NP**-complete. Given the (easy) observation that `BH` is **NP**-complete (Lemma 3.4), one understand that to prove the **NP**-completeness of `Circuit-SAT` it suffices to simulate Turing machines by Boolean circuits. The following lemma will help us in this business. (In our application the machine will be polynomial-time, so the number of steps will be polynomial.)

Lemma 10.1. *Let M be a one-tape polynomial-time DTM. Then, given the "input size" n the "number of steps" $t \geqslant n$, one can construct in time polynomial in n and t, a Boolean circuit C such that* $\forall x \in \{0,1\}^n \ (C(x) = 1 \iff M \textit{ accepts } x \textit{ in at most } t \textit{ steps})$.

Proof. We will simulate the work of M in stages. At each stage i we will compute the configuration of M after the i-th step. The i-th configuration will consist of

- The state of M ($O(1)$ bits, call them $s_{i,1}, \dots, s_{i,b}$),
- The memory of M ($O(t)$ bits, because it has no time to use more, call them $m_{i,1,1}, \dots, m_{i,t,c}$, where c is [the logarithm of] the tape alphabet size of M),
- The head position of M (in principle, one could encode it using $O(\log t)$ bits, but we will do it inefficiently using Boolean variables $h_{i,1}, \dots, h_{i,t}$, only one of them will be true for each value of i).

Thus we will design a subcircuit C_i with inputs for the $(i-1)$-th configuration and outputs for the i-th configuration. We will then connect the outputs of the previous stage subcircuit to the inputs of the next stage subcircuit thus connecting, in total, t subcircuits. The input of the first subcircuit is the starting configuration, which contains the variables (input bits

for C) for the machine's input x and obvious constants (such as the encoding of q_S and the initial position of the head). The outputs of the last subcircuit, namely $s_{t,1}, \ldots, s_{t,b}$, will be fed to a simple circuit checking whether the resulting state is q_Y, and this result will be the only Boolean output of C. (W.l.o.g. one can assume that once a machine comes to an accepting state, it remains in it forever without doing anything.)

Observe that most values $m_{i,j,k}$ (and $h_{i,j}$) can be just copied from $m_{i-1,j,k}$ (and $h_{i-1,j}$, respectively). Only the values in the neighbourhood of the head (i.e., where $h_{i-1,j} = 1$) can change. This change can be computed from the transition function of M; this is a function with $O(1)$ inputs (the contents $m_{i-1,j\pm1,k}$ of the neighbouring cells for every k, the state bits $s_{i-1,\ell}$ for every ℓ, and also $h_{i-1,j\pm1}$). One can compute each bit of its result as a Boolean circuit (or even a formula in CNF) by inspecting the truth table (which is exponential size — but this is an exponent in a constant number of inputs, thus a constant as well). This truth table is determined by the transition function of M.

Therefore, all the bits of the next configuration can be computed by constant-size multi-output subcircuits $C_{i,j}$ for $1 \leqslant j \leqslant t$. To complete the construction of C_i it remains to compute the next state of M. Again, one can compute its r-th bit as a constant-size subcircuit $C'_{i,j,r}$ conditioned on the fact that the head was in the position j. Then $C'_{i,r} = \bigvee_{j=1}^{t}(C'_{i,j,r} \wedge h_{i,j})$, let us call C'_i the multi-outpu union of these circuits for every r.

The final construction of the subcircuit C_i computing the i-th configuration from the $(i-1)$-th configuration includes small subcircuits $C_{i,j}$ for every position j as well as C'_i.

We illustrate this construction in Figire 1. □

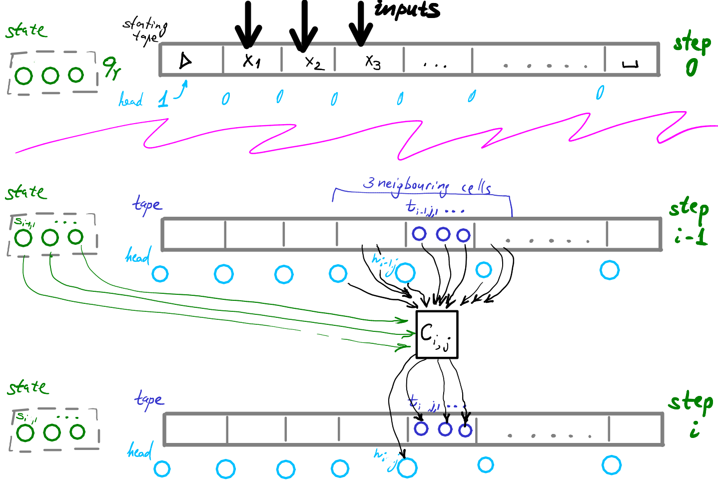


Figure 1: The inputs to C. The circuit $C_{i,j}$.

Remark 10.1. One can observe that if M is NTM, then instead of computing the next configuration one can at least construct a circuit that given all the configuration on input will check whether the sequence of configurations is a correct run of a nondeterministic

machine (for some nondeterministic choices). One can construct even a Boolean formula by connecting the checking circuits D_i for every step i in a big conjunction.

Remark 10.2. Note that the circuits constructed in Lemma 10.1 and Remark 10.1 are very simple and standard. In order to print such a circuit, one needs just a constant number of "for" cycles for cycle variables denoting the step number, the cell number, and a couple of similar counters. Thus one can print such a circuit using just $O(\log t)$ additional memory cells — we will formulate it rigorously and use this fact in the end of the course when we will discuss space complexity.

With Lemma 10.1 at hand, we can now prove the **NP**-completeness (and even more) of `Circuit-SAT` and then use it to complete the proof of the Cook–Levin theorem.

Lemma 10.2. `Circuit-SAT` *is* **NP**-*complete. Moreover, for every* $L \in \mathbf{NP}$ *defined by a polynomially bounded relation computed by a polynomial-time DTM* M *as* $L = \{x : \exists w\ M(\langle x, w\rangle) = 1\}$ *there exists a polynomial-time many-one reduction* f *that on input* x *outputs a circuit* D *such that* $\forall w\ \ (M(\langle x, w\rangle) = 1 \iff D(x, w) = 1)$.

Proof. The fact that `Circuit-SAT` $\in \mathbf{NP}$ is obvious (we can compute the value of a circuit on a given input by computing the values coputed at its gates in a topological order).

Note that the fact that in the statement the equivalence holds $\forall w$ implies that f is a correct reduction: it would be enough for it even to have that $(\exists w\ M(\langle x, w\rangle) = 1) \iff (\exists w\ D(x, w) = 1)$.

Let us compute the circuit C by Lemma 10.1. It has inputs both for x and w (don't get lost in the notation: x in this lemma and x in Lemma 10.1 are instances of different problems). Let us hardwire the constants — the bits of x — into C; denote the resulting circuit as C_x, its only variables are the bits of w. By Lemma 10.1, $C_x(w) = 1$ if and only if $M(\langle x, w\rangle) = 1$. □

Remark 10.3. Note that Lemma 10.2 allows to finish also the first step of the search-to-decision reduction in Theorem 9.2, because the circuit that we constructed has exactly the same solutions as the search problem associated with M.

We are ready to finalize the proof of the Cook–Levin theorem:

Proof of Theorem 10.1. It is obvious that `3-SAT` $\in \mathbf{NP}$. To prove its hardness, use the **NP**-completeness of `Circuit-SAT` (Lemma 10.2) and then reduce `Circuit-SAT` to `3-SAT` (Lemma 3.3). □

11 The Polynomial Hierarchy

Previously we studied the Arithmetical Hierarchy, which was characterized by sequences of quantifiers with at most k quantifier changes, and a recursive predicate at the end. We now come to the Polynomial Hierarchy, which is very similar to it, but now everything will be polynomially bounded (balanced) and polynomial time.

Similarly to the Arithmetical Hierarchy, the classes of the Polynomial Hierarchy can be defined both using a k-tower of oracle computations (now based on **NP** instead of **RE**) and using k [sequences of] quantifiers. Spoiler: later on we will study polynomial-time computations unbounded number of quantifiers and they will correspond to (many-round) two-player games! (Like [infinite] chess.) And also — surprisingly — to computations using a polynomial amount of space.

An example of a problem in (the second level of) the Polynomial Hierarchy is the following language:

Example 11.1.
Input: a Boolean circuit C with $2n$ bits of inputs and one bit of output.
Output: say whether $\exists x\ \forall y \neq x\ C(x,y) = 1$.
(Here $x, y \in \{0,1\}^n$.)

(Motivation: $C(x,y)$ says whether the team x wins its home field game over the team y. The question is whether there exists a team that wins all its games at home.)

Let us give a formal definition (even two) to such classes.

11.1 Two equivalent definitions

Definition 11.1 (classes of the polynomial hierarchy)**.**

$$
\begin{aligned}
\Sigma_0^p &= \Pi_0^p = \Delta_0^p = \mathbf{P} \\[1ex]
\Sigma_1^p &= \mathbf{NP} \\
\Pi_1^p &= \mathbf{co\text{-}NP} \\
\Delta_1^p &= \mathbf{P} \\[1ex]
\Sigma_k^p &= \mathbf{NP}^{\Pi_{k-1}^p} = \mathbf{NP}^{\Sigma_{k-1}^p} \\
\Pi_k^p &= \mathbf{co\text{-}}\Sigma_k^p \\
\Delta_k^p &= \mathbf{P}^{\Sigma_{k-1}^p}
\end{aligned}
$$

An alternative definition stems from the following theorem:

Theorem 11.1. $\forall k \geqslant 1$

1. *$L \in \Sigma_k^p \iff$ there is a polynomial p and a binary relation $T \in \bigcup_{n\in\mathbb{N}} \{0,1\}^n \times \{0,1\}^{\leqslant p(n)}$ such that $T \in \Pi_{k-1}^p$ and $\forall x$*

   $$x \in L \iff \exists w\ \langle x, w\rangle \in T,$$

2. $L \in \Pi_k^p \iff$ *there is a polynomial p and a binary relation $S \in \bigcup_{n\in\mathbb{N}} \{0,1\}^n \times \{0,1\}^{\leqslant p(n)}$ such that $S \in \Sigma_{k-1}^p$ and $\forall x$*

$$x \in L \iff \forall w \ \langle x, w\rangle \in S$$

By applying the two parts of this theorem repeatedly, one comes to the following corollary:

Corollary 11.1. $\forall k \geqslant 1$

1. $L \in \Sigma_k^p \iff$ *there is a polynomial p and a $(k+1)$-ary polynomial-time decidable relation $S \in \bigcup_{n\in\mathbb{N}} \{0,1\}^n \times \{0,1\}^{\leqslant p(n)} \ldots \times \{0,1\}^{\leqslant p(n)}$ such that $\forall x$*

   $$x \in L \iff \exists w_1 \forall w_2 \exists w_3 \ldots Q w_k \ \langle x, w_1, w_2, \ldots, w_k\rangle \in R,$$

   where $Q = \exists$ if k is odd
   and $Q = \forall$ if k is even.

2. $L \in \Pi_k^p \iff$ *there is a polynomial p and a $(k+1)$-ary polynomial-time decidable relation $R \in \bigcup_{n\in\mathbb{N}} \{0,1\}^n \times \{0,1\}^{\leqslant p(n)} \ldots \times \{0,1\}^{\leqslant p(n)}$ such that $\forall x$*

   $$x \in L \iff \forall w_1 \exists w_2 \forall w_3 \ldots Q w_k \ \langle x, w_1, w_2, \ldots, w_k\rangle \in R,$$

   where $Q = \exists$ if k is even
   and $Q = \forall$ if k is odd.

Note that even if we prove the theorem only up to a certain level k, we can prove the corollary for this level k. We will use this idea in our (inductive) proof of the theorem.

Proof of Theorem 11.1. It suffices to prove item 1, because they item 2 is symmetric to it ($\Pi_k^p = \mathbf{co}\text{-}\Sigma_k^p$ by definition, so just apply the negation to the characterization of L properly).

Base: for $k = 1$ this is the alternative definition of $\Sigma_1^p = \mathbf{NP}$: Definition 1.2 says that $L \in \mathbf{NP} \iff$ there is $T \in \mathbf{P}$ s.t. $L = \{x : \exists w \ \langle x, w\rangle \in T\}$.

Step $(k-1 \mapsto k)$:

$\boxed{\Rightarrow}$: $L \in \Sigma_k^p = \mathbf{NP}^{\Sigma_{k-1}^p}$, therefore there is an oracle polynomial-time NTM $N^\bullet$ and an oracle $O \in \Sigma_{k-1}^p$ s.t. $L = L(N^O)$. By induction hypothesis there is $T' \in \Pi_{k-2}^p$ such that $O = \{q : \exists w \ \langle q, w\rangle \in T'\}$.

To decide $x \in^? L$, we need to run N^O on it, but how? We do not have O. However, we have the existential quantifier. Thus let us ask for a witness $(\exists w_1)$ and check it. The witness must contain, at least,

(1) The accepting path of N^O. (That is, the sequence of nondeterministic choices or even the sequence of configurations of N leading to an accepting configuration.)

However, while in most cases we can easily check that $N^\bullet$ indeed behaves like written in this witness (that is, we can simulate its behaviour with these nondeterministic choices or just check the validity of each step if the configurations are given in full), we cannot check the steps where $N^\bullet$ comes to the special oracle state. Therefore, we need more information to be included in the witness:

(2) The answer a_t of the oracle O for each query q_t of N^O.

Now simulating (or checking) the accepting path of N^O is easy, but checking without O that (2) indeed lists *correct* answer seems impossible unless we have more info in the witness. Thus let us ask for it:

(3) For each t s.t. $a_t = 1$ the witness includes $\boldsymbol{w} = w^{(t)}$ for $q_t \in O$ (according to the definition of O via $\exists w$ and T').

Thus it remains to check this last part (everything else can be then checked in polynomial time).

In total, our witness verification will look like that:

"Program" for $T \in \Pi_{k-1}^p$ s.t. $L = \{x : \exists w_1 \ \langle x, w_1 \rangle \in T\}$:

Witness: an accepting path of N^O, oracle answers a_t, witnesses $w^{(t)}$.

1. Simulate N^O on the accepting path using a_i's as the oracle answers. // *0 quantifiers*
2. For every $a_t = 1$, check this answer using $w^{(t)}$ and T': $\langle q_t, w^{(t)}, w_3, \ldots \rangle \in T'$. // $T' \in \Pi_{k-2}^p$
3. For every $a_t = 0$, check this answer using $\overline{O}$. // $\overline{O} \in \Pi_{k-1}^p$

We need to show that this "program" can be implemented as a Π_{k-1}^p computation. Indeed, it is a conjunction of several Π_{k-1}^p computations (also of Π_{k-2}^p, but this is a particular case), because it accepts if all the checks are OK.

NB!!! *It is important that it is a conjunction. One cannot combine such computations arbitrarily, it would give a* $\Delta_k^p = \mathbf{P}^{\Pi_{k-1}^p}$ *computation.*

How can we merge many Π_{k-1}^p computations into one? We will give two versions of the proof that it is possible. The first version is very formal (using manipulations with logical formulas and quantifiers), the second version is more intuitive and uses computation trees.

Version 1 (using logic): Apply the induction hypothesis iteratively to $\overline{O}$:

$\overline{O} = \{q : \forall w_2 \exists w_3 \ldots Q w_k \ \langle q, w_2, w_3, \ldots, w_k \rangle \in R'\}$ for some polynomially bounded $R' \in \mathbf{P}$.

One can write a similar formula for T' as well. Our "program" makes several queries to $\overline{O}$ and T', but observe that for different queries q_t the quantified variables in this definition are different (the variables in w_i for q_j have nothing to do with variables in $w_{i'}$ for $q_{j'}$ for $j \neq j'$).

Recall from the mathematical logic course that one can move quantifiers through independent variables like in $\forall u\ A(z,u)\ \wedge\ \forall v\ B(z,v)\quad\equiv\quad \forall u \forall v\ (A(z,u) \wedge B(z,v))$.

So let us take the conjunction of all the formulas for all the queries to $\overline{O}$ and T', and move all the quantifiers to the front. We thus get a formula $\forall w_2^{(1)}, w_2^{(2)}, \ldots \exists w_3^{(1)}, w_3^{(2)}, \ldots$ with a polynomial-time decidable predicate at the end that verifies that all the checks are OK (from the quantifiers it has the values of all these variables $w_i^{(j)}$ and also checks the first line of our "program", thus it knows the queries as well).

Version 2 (using computation trees):

Consider two languages $A, B \in \Pi_{k-1}^p$, that is, two conondeterministic machines N_A^Q, N_B^Q with an oracle $Q \in \Sigma_{k-2}^p$.

We want to compute a conjunction, that is, the intersection of these languages ($A \cap B$). Let us start N_A^Q. It works nondeterministically. If it rejects on a particular path, we reject. If it accepts, we start N_B^Q (on the same input) and accept or reject like it does.

If N_A^Q had at least one rejecting path, the resulting machine will reject.

If N_B^Q had at least one rejecting path, the resulting machine will reject as well.

Only if both machines have only accepting paths, the resulting machine accepts. Thus we designed a conondeterministic machine N_C^Q that uses the same oracle and accepts $A \cap B$. (See Fig. 2 for illustration.)

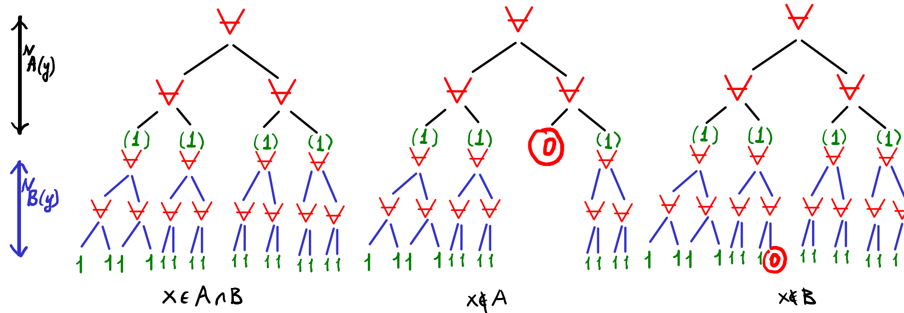


Figure 2: A conjunction of two conondeterministic computations.

In our proof we need to combine a polynomial number of runs of conondeterministic machines for $\overline{O}$ and T', thus the computation tree will be composed of a polynomial number of levels of computation trees inserted into each other, which is OK.

The nice (and very useful) intuition stops here, technical details remain. Namely, the only small problem is that T' is using a different oracle than $\overline{O}$ (though from a lower class). One can solve this problem, *for example,* by considering a single oracle (language) D that merges queries to two different oracles (languages) D_0, D_1:

$$D = \{bz : b \in \{0,1\}, z \in \{0,1\}^n, z \in D_b\},$$

one can use this language to check $z \in D_0$ and to check $z \in D_1$. This language performs essentially a single computation (either D_0 or D_1), in our case the oracle in it is again one of two oracles — yet of a lower level class, thus it can be eventually traced to the level 0.

$\boxed{\Leftarrow}$ This direction is easy: the nondeterministic oracle machine will use T as an oracle. It will guess w, query $\langle x, w \rangle \in T$, and return this answer. $\square$

11.2 Complete problems for the classes of the Polynomial Hierarchy

Let us exhibit a complete problem for Σ_k^p (its complement is that complete for Π_k^p). Note that for $k = 1$ this is SAT (we do not need to mention $\exists w_1$ — it is assumed in the formulation of SAT), for $k > 1$ we need to write down the quantifiers explicitly in order to state which Boolean variables correspond to which quantifiers.

Theorem 11.2. $\mathtt{QBF}_k = \{$ *True Boolean formulas of type* $\exists w_1 \forall w_2 \dots Q w_k\ \Phi(x, w_1, \dots, w_k)\}$ *is* Σ_k^p*-complete, where* Φ *is in CNF if* $Q = \exists$ *, and* Φ *is in DNF if* $Q = \forall$*.*

Proof. First of all, $\mathtt{QBF}_k \in \Sigma_k^p$ by Corollary 11.1. Let us prove that it is hard for Σ_k^p. Consider any $L \in \Sigma_k^p$ and construct a reduction that will map an input x for the decision problem $x \in^? L$ into a quantified formula.

We will first do with a Boolean circuit instead of a CNF/DNF, then we will convert it into a formula.

Since $L \in \Sigma_k^p$, by Corollary 11.1 $L = \{x : \exists w_1 \forall w_2 \dots \exists w_k\ \langle x, w_1, \dots, w_k \rangle \in R\}$, where R is decided by some DTM working in polynomial time $p(n)$.

Our reduction will do the following on input x_* of length n:

- build a circuit C_n for the $p(n)$-time run of R on input x (recall the proof of the **NP**-completeness of Circuit-SAT, namely Lemma 10.1), this circuit can be built in polynomial time and will contain variables for the bits of $x, w_1, \dots, w_k$.

- hardwire the values for $x = x_*$ into this circuit (that is, replace them by the corresponding constants that we see in our input) and write down the expression

$$\exists w_1 \forall w_2 \dots \exists w_k\ C_n|_{x=x^*}(w_1, \dots, w_k)$$

  (we wrote it as $C \dots (\dots)$ to emphasize that $w_1, \dots, w_k$ remain variables).

It would be enough to output this expression for a "circuit version" of $\mathtt{QBF}_k$. Now let us convert this circuit into a formula in CNF or DNF. Recall the reduction f from Circuit-SAT to 3-SAT (Lemma 3.3). In the proof we have shown the equivalence (3), that is, $f(C) = \Phi$ (a formula in 3-CNF) such that $\forall a\ (C(a) = 1 \iff \exists b\ \Phi(a, b) = 1)$. Thus we can equivalently replace the circuit $C = C_n|_{x=x^*}$ with variables in $a = (w_1, \dots, w_k)$ by the so-constructed $\exists b\ \Phi$ with variables in a, b.

Thus if the last quantifier $\exists w_k$ is the existential one, the quantifiers $\exists w_k \exists b$ merge into a single existential quantifier, and we are done in the case of odd k.

If k is even, the last quantifier $\forall w_k$ is the universal one; in this case let us apply Lemma 3.3) to $\neg C_n|_{x=x^*}(w_1, \ldots, w_k)$ instead and get a formula Ψ in 3-CNF such that $\neg C_n|_{x=x^*} = 1 \iff \exists b\ \Psi = 1$. The equivalent formula for the original circuit $C_n|_{x=x^*}$ will be thus $\forall b\ \overline{\Psi}$, that is, it will be in 3-DNF (observe that the negation and de Morgan laws turn a CNF into a DNF). Therefore, the quantifiers $\forall w_k \forall b$ merge into a single universal quantifier, and we are done in the case of even k as well. □

11.3 PH altogether

The union of all the classes of the polynomial hierarchy is denoted

$$\mathbf{PH} = \bigcup_{k \in \mathbb{N}} \Sigma_k^p.$$

The picture (see Fig. 3) for the polynomial hierarchy is similar to the one for the arithmetical hierarchy, yet we do not know whether $\mathbf{NP} = \mathbf{co\text{-}NP}$ or $\mathbf{P} = \mathbf{NP} \cap \mathbf{co\text{-}NP}$, and a similar question remains open for higher levels of the hierarchy as well.

11.4 Collapsing the Polynomial Hierarchy

In complexity theory it is a custom to prove theorems modulo some conjectures (open problems). In particular, when we cannot prove something conditioned on $\mathbf{P} \neq \mathbf{NP}$, we sometimes prove it conditioned on a stronger statement: the absence of a *collapse of the polynomial hierarchy*. To prepare for it, let us make several easy observations, which we formulate for Σ_k^p classes, but similar ones are valid for Π_k^p classes by symmetry.

Observation 0. All these classes are closed under many-one polynomial-time reductions. (Indeed, we can incorporate the reduction into the main (lower) machine):

If $L \to^p L'$ and $L' \in \Sigma_k^p$, then $L \in \Sigma_k^p$.

Observation 1. If $\mathbf{P} = \mathbf{NP}$, then all these classes coincide. (Because the hierarchy would be made using polynomial-time decidable oracles: $\Sigma_2^p = \mathbf{NP}^{\mathbf{NP}} = \mathbf{NP}^{\mathbf{P}} = \mathbf{P}$, etc.)

Observation 2. If for some specific k it happens that $\Sigma_k^p = \Pi_k^p$, then the hierarchy stops growing at this level k: $\Sigma_k = \Sigma_{k+1} = \ldots$.

Proof. Recall Theorem 11.1 (and Corollary 11.1). A **language in** Σ_{k+1}^p is characterized as $\{x : \exists w_1\ \langle x, w_1 \rangle \in T$ for $T \in \Pi_k^p$. Under our assumption $T \in \Sigma_k^p$, thus T is characterized as $\{y : \exists w_2\ \langle y, w_1 \rangle \in S\}$ for $S \in \Pi_{k-1}^p$. Thus the original language is characterized as $\{x : \exists w_1 \exists w_2 \langle \langle x, w_1 \rangle, w_2 \rangle \in S\}$; the two existential quantifiers then merge into a single quantifier and this characterization satisfies the **definition of** Σ_k^p. □

This situation is called the collapse of the hierarchy. Even though we cannot prove it does not happen, we do not believe in such collapses. Here is a typical statement conditioned on the absence of a collapse:

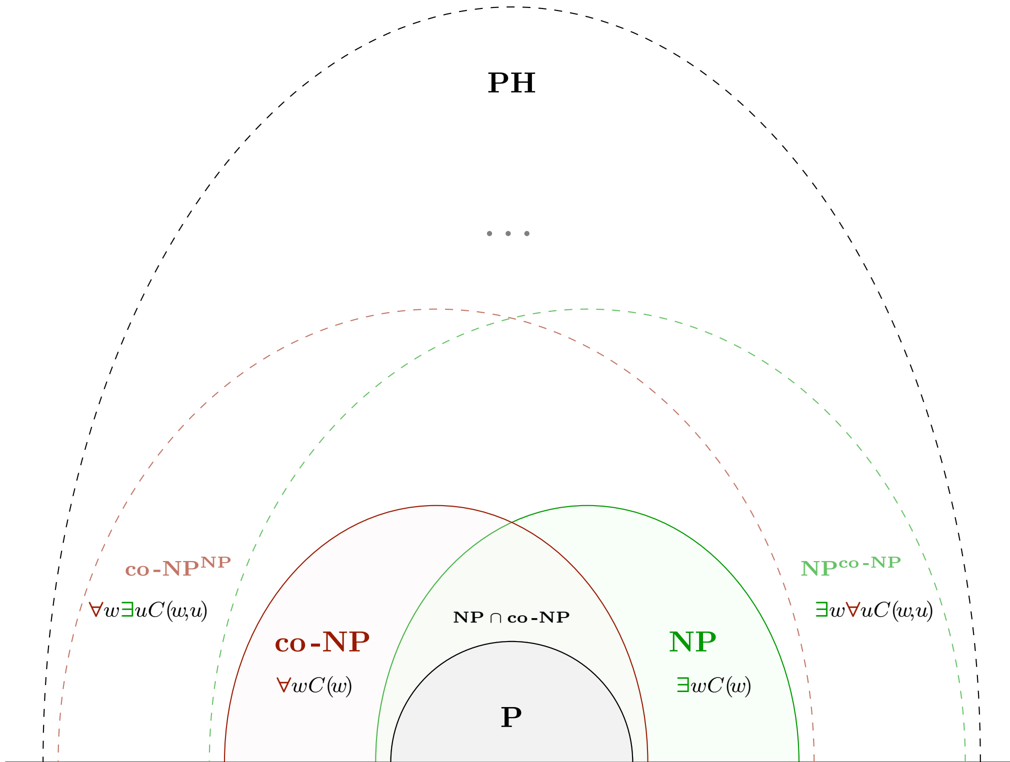


Figure 3: Overview of the classes of the polynomial hierarchy.

Observation 3. The class **PH** has no complete languages *unless the polynomial hierarchy collapses*.

Proof. If some language L' is complete for **PH**, then $L' \in \Sigma_k^p$ for some specific k, then every $L \in \mathbf{PH}$ reduces to L', thus (by Observation 0) $L \in \Sigma_k^p$ as well.

Therefore, $\mathbf{PH} \subseteq \Sigma_k^p$. □

12 Polynomial space: The class PSPACE

12.1 A motivating example

Let us consider a typical two-player game.

Example 12.1 (GEOGRAPHY game)**.** The input to this problem is a **directed graph** $G = (V, E)$ and a **starting node** $s \in V$. Two players are building a directed simple path starting from s ("simple" means that it does not come to the same node twice). The first player selects an edge $(s, x) \in E$ from s, then the second player selects an edge $(x, y) \in E$ coming from the end x of the previous edge, etc. The one who cannot move a legal move (every available edge goes to an already visited node) loses; the other one wins.

The computational problem associated with this game is:

Given G and s, say whether the first player has a winning strategy.

In simple words, $\exists$ a move of the first player such that $\forall$ move of the second player $\exists$ a move of the first player such that $\forall$ ... (etc) ... $\exists$ a move of the first player such that the second player has no legal move. □

This is an example of a finite game with two outcomes (either the first player or the second player wins)! Therefore, for given G and s either the first or the second player has a winning strategy.

Caution!!! Never apply this kind of logic to *infinite* games even if they appear to have only two outcomes!

This game is an example of a problem of the class **PSPACE**: even though it may require exponential time, one can prove that it can be decided using only a polynomial amount of memory.

Note important features of this game:

- At each step, the range of moves is limited (by a polynomial).
- The length of this game is limited (by a polynomial).
- Both "the set of valid moves" and "whether this position is winning" can be checked in polynomial time.

Because of them one can design a recursive algorithm — depth-first search with polynomial (due to item 2) depth — that will use polynomial space.

Exercise 12.1. Complete a formal proof that the existence of a winning strategy for this game can be checked deterministically within polynomial space. It may be easier to do it after in what follows we consider a recursive polynomial-space algorithm for `QBF`.

In fact, **PSPACE** is essentially the class of such two-player games, as we will see later.

12.2 Space complexity classes — a formal definition

Recall that time is the number of steps before a Turing machine stops. Space is the number of cells it uses, that is, the sum of the rightmost positions of its heads during the run of the machine.

When we consider severe space restrictions (smaller than the input size), it becomes important that the input tape must be separated and made readonly (and its head position is not counted in the space used), but in this lecture we consider polynomial space restriction — and for polynomial space it does not matter.

Let us define formally space complexity classes:

$$\mathbf{DSpace}[f(n)] = \{L \mid L \text{ is decided by a DTM using space } O(f(n))\}.$$

We will consider only reasonable ("space-constructible") functions f, that is, $f(n)$ must be non-decreasing and computable within space $O(f(n))$ on input 1^n.

$$\mathbf{PSPACE} = \bigcup_{k\geq 0} \mathbf{DSpace}[n^k].$$

For a NTM (considered as a machine that really makes nondeterministic choice, not with a witness definition), the space is counted as the maximum space used on all nondeterministic paths.

$$\mathbf{NSpace}[f(n)] = \{L \mid L \text{ is decided by a NTM using space } O(f(n))\}.$$

One can define

$$\mathbf{NPSPACE} = \bigcup_{k\geq 0} \mathbf{NSpace}[n^k],$$

but very soon we will see that it equals just **PSPACE**.

A "standard" example of a **PSPACE** (and even **PSPACE**-complete problem) is a space-bounded version of the Bounding Halting (Bounded Acceptance) problem:

Example 12.2 (`SPACE-BH`, `SPACE TMSAT` in the book)**.**

$$\texttt{SPACE-BH} = \{\langle M, z, 1^t\rangle : \text{ DTM } M \text{ accepts } z \text{ using space at most } t\}.$$

Why it can be computed within polynomial space? Recall from the construction of UTM that if $M(x)$ uses space $s(|x|)$, then $\mathrm{UTM}(\langle M, x\rangle)$ can use space $O(s(|x|))$.

An easy exercise (straightforward from the definitions) is:

Exercise 12.2. Show that `SPACE-BH` is **PSPACE**-complete under polynomial-time many-one reductions.

12.3 Quantified Boolean formulas — a PSPACE-complete problem

We consider **PSPACE**-hardness and **PSPACE**-completeness under polynomial-time reductions. Note that **PSPACE** is closed under them. Note also that many smaller classes (such as **P**, **NP**, **PH**) are closed under them, thus proving that a **PSPACE**-hard problem belongs to one of these classes will put the whole class **PSPACE** into the corresponding class.

We will show that the following language `QBF` (also known as `QBF-SAT` or `TQBF`) is **PSPACE**-complete.

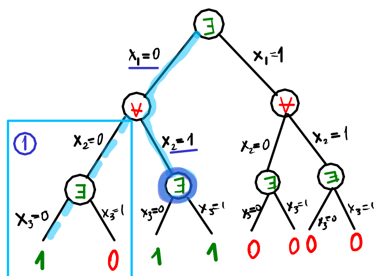
Definition 12.1. `QBF` = { true quantified CNF formulas $Q_1x_1\ Q_2x_2 \dots\ Q_mx_m\ \Phi$ }, variables x_i denote single bits, $Q_i \in \{\forall, \exists\}$, where Φ is a Boolean formula in CNF mentioning the variables $x_1, x_2, \dots, x_m$ and only them (logicians call it a "closed formula").

Note that this definition generalizes $\texttt{QBF}_k$ we have earlier seen (even though x_i's here are bits and not strings, since the number of quantifiers is unlimited, $\texttt{QBF}_k$ remains a particular case of `QBF`). Therefore, the fact that `QBF` $\in$ **PSPACE** proves that **PH** $\in$ **PSPACE**.

An easy exercise: for `QBF`, it does not matter whether the formula is in CNF, DNF, or even if it is a Boolean circuit and not a formula: all these problems are polynomial-time equivalent.

Theorem 12.1. `QBF` *is* **PSPACE**-*complete.*

Proof.
`QBF` $\in$ **PSPACE**: Conside Φ's brute-force search tree (a leaf $0, 1, 0$ gets the value $\Phi(0, 1, 0)$):



To compute the value of $Q_1x_1\ Q_2x_2\ \dots Q_mx_m\ \Phi(x_1, x_2, \dots, x_m)$, one needs to consider two values $x_1 = 0$ and $x_1 = 1$, compute the values of the subformulas (corresponding to the two subtrees) $Q_2x_2\ \dots Q_mx_m\ \Phi(0, x_2, \dots, x_m)$ and $Q_2x_2\ \dots Q_mx_m\ \Phi(1, x_2, \dots, x_m)$ and return the maximum of these values if $Q_1 = \exists$ (respectively, the minimum if $Q_1 = \forall$). This suggests a recursive procedure that removes quantifiers until the values of all variables are known; then it can return the value by evaluating Φ on them, it is a leaf of our recursion tree.

So we do it recursively using depth-first search. For this search and similar tasks we need to keep in our memory the current path (of length at most m) and the two last values (the value of the left subtree computed by a recursive call and the value of the right subtree computed by a recursive call). If the quantifier in a node is $\exists$, the recursive calls return max of the values of its two subtrees; otherwise (if it is $\forall$) it returns the min. However, since we

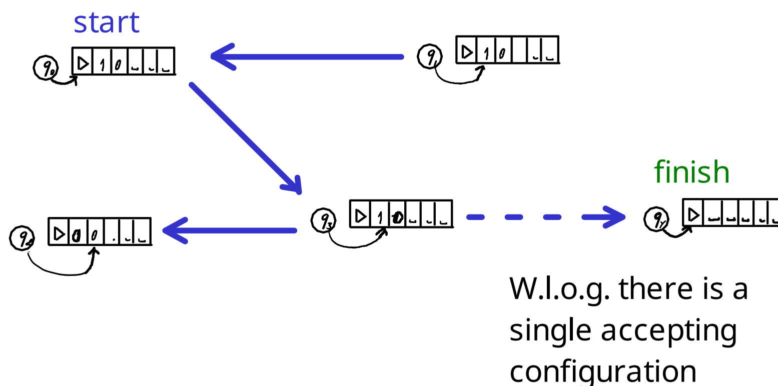
are dealing with quantifiers, we can save a bit of time (this is not important, but software engineers would be frustrated if we don't do it):
`Recursive procedure Eval (input: node` v`).`

1. `If` v `is a leaf (an assignment for all the variables), return the value of` Φ `in it.` *// This is our recursion base.*
2. `Let` v_1, v_2 `be the two children of` v `(that is, the two subtrees for the two values of the next unassigned variable). Then`
   (a) `Make a recursive call` $e_1 :=$ `Eval(`v_1`).`
   (b) `If` $e_1 = 1$ `and the quantifier in` v `is` $\exists$`, then return 1.`
   (c) `If` $e_1 = 0$ `and the quantifier in` v `is` $\forall$`, then return 0.`
   (d) `Make a recursive call Eval(`v_2`) and return its result.`

`QBF` is **PSPACE**-hard: Consider an arbitrary language $L \in$ **PSPACE**, let us reduce it to `QBF`. So we are given some input x and will transform it into a quantified Boolean formula.

This language is accepted by some polynomial-space Turing machine M. (We can assume it is a 1-tape machine.) A configuration of this machine contains its memory, its state, and the position of the head. Thus it can be described by polynomially many bits (call it $p(n)$), so the number of possible configurations is $2^{p(n)}$.

We are interested in the Reachability problem for this configuration graph. Namely, we are interested in the existence of a directed path from the starting configuration c_{start} (containing this particular input x) to the accepting configuration c_{accept} (note that we can assume that a Turing machine has exactly one accepting configuration, because we can force it to "tidy up" just before accepting: when it is ready to come to the accepting state q_Y, it will wipe the tape and return the head to the leftmost position — and then accept).



We will build inductively a formula that computes the following predicate:

$$\Phi_i(c_1, c_2) = \text{«there is a path } c_1 \rightsquigarrow c_2 \text{ of length} \leqslant 2^i\text{»}.$$

Then $\Phi_{p(n)}(c_{start}, c_{accept})$ will be the result of our reduction: it says whether there is an accepting computation of length at most $2^{p(n)}$ (which is the number of vertices in the graph),

but we are not interested in longer computations as some configuration would be necessarily repeated repeated on such a long path.

Once could try to express Φ_i as

$$\Phi_i(c_1, c_2) \ = \ \exists \boldsymbol{d} \ \ \Phi_{i-1}(c_1, \boldsymbol{d}) \wedge \Phi_{i-1}(\boldsymbol{d}, c_2). \tag{8}$$

It literally means that there is a path $c_1 \rightsquigarrow c_2$ of length $\leqslant 2^i$ if and only if there is an intermediate node d on this path such that there are paths $c_1 \rightsquigarrow d$ and $d \rightsquigarrow c_2$ of length twice as small.

However, this definition of Φ_i is not efficient: if one would unroll it inductively (replacing Φ_{i-1} by its definition, etc), then there an exponential blowup would occur, because there are two occurrences of Φ_{i-1} in this expression.

Thus let us use a more efficient expression:

$$\boldsymbol{\Phi_i}(c_1, c_2) = \exists \boldsymbol{d} \forall x \forall y (((x = c_1 \wedge y = \boldsymbol{d}) \vee (x = \boldsymbol{d} \wedge y = c_2)) \Rightarrow \boldsymbol{\Phi_{i-1}}(x, y)).$$

Now there is only one occurrence of Φ_{i-1}, and if we substitute these expressions until we reach the induction base Φ_0, the size of the resulting formula will be bounded by a polynomial.

The induction base $\Phi_0(c_1, c_2)$ checks whether $c_1 = c_2$ or the transition from c_1 to c_2 is a legal step of the Turing machine M. Both of these conditions are, of course, polynomial-time computations; like in the proof of the Cook–Levin theorem, we can write down a polynomial-size Boolean circuit or even a Boolean formula with existential quantifiers that checks this fact. (Namely, checking the transition is like in the Cook–Levin theorem itself, we even do not produce c_2 from c_1 and only check that the given transition is correct; checking $c_1 = c_2$ is even easier as this is an AND of XNORs: $c_1 = c_2 \iff \displaystyle\bigwedge_{j=1}^{p(n)} (c_1[j] \oplus c_2[j] \oplus 1) = 1$.)

After we finish the construction, we move all the quantifiers to the front and convert the remaining part of the formula into a CNF (with existential quantifiers) using Tseitin's reduction (Lemma 3.3). The result is the output of our reduction: it is a quantified Boolean formula that is true if and only if there is a path from the starting to the accepting configuration in the configuration graph of $M(x)$.

(Check your understanding: what are the variables of this formula? Spoiler is in the footnote[12].) ☐

Corollary 12.1. PSPACE = NPSPACE, *and thus* **NPSPACE = co -NPSPACE**.

Proof. This is a corollary of the proof and not the statement of the theorem. Note that we did not use the fact that the machine was deterministic: we simply checked the *existence* of

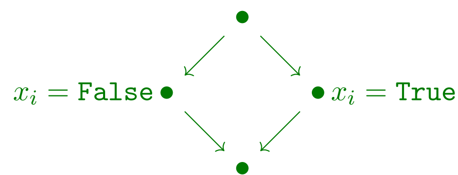
[12] Answer: They are variables introduced by the quantifiers $\exists d \ \forall x \ \forall y$ at each stage of the construction. Each such quantifier generates $p(n)$ Boolean variables. Additionally, there are variables from converting a polynomial-time computation to a CNF via a Boolean circuit. However, c_{start} and c_{accept} are not variables, they are constants: c_{accept} contains q_Y, the empty tape $\triangleright$, and the head position 0, and c_{start} contains $\triangleright x$ with enough space after the input (x is not a variable: its bits are constants given to our reduction on the input), q_S, and the head position 0).

a path from the starting to the accepting configuration. If we deal with the configuration graph of an NTM, it does not change anything, it still contains just exponentially many configurations, each one of polynomial size. □

Corollary 12.2. PH ⊆ PSPACE.

Proof. As mentioned above, $\texttt{QBF}_k$ is a particular case of $\texttt{QBF}$. □

Exercise 12.3. Prove that the game we presented in the beginning of the section (Example 12.1) is **PSPACE**-complete. **Hint:** reduce $\texttt{QBF}_3$ to it, and consider it as a game between ∃-player and ∀-player choosing values for the respective variables one by one. The key gadget (building block of a graph) is:



Remark 12.1. One can prove Corollary 12.1 easier by looking at the equation (8) from a software engineering point of view (or using the same idea of depth-first search that we used in proving $\texttt{QBF} \in$ **PSPACE**). This equation defines a simple recursive function

$\Phi(c_1, c_2, i)$:

“`if` $i \leqslant 1$ `then [return the answer non-recursively];`

`for` $d := 1$ *to* N `do`
  `if` $\Phi(c_1, d, i-1)$ `and` $\Phi(d, c_2, i-1)$ `then return True;`
`return False.`

Here N is the number of nodes in the graph; c_1, c_2, i, d consist of $\lceil \log_2 N \rceil + 1$ bits each. As we know from software engineering, the local data of a recursive call (the arguments, local variables, the return value) is put on the stack and is removed from the stack after return. Our function does not modify any global data (does not use the “heap”). Therefore, the space used by this recursion is the depth of recursion times the size of local data, i.e., $O((\log_2 N)^2)$ (that is, $O(p(n)^2)$ for our particular graph of configurations). They key idea is that after the return, all the data of the recursive call is dropped and the space can be reused.

12.4 PSPACE on our map

Our current understanding is given in Fig. 4. Let us consider two containments shown on it that we have not yet discussed in full.

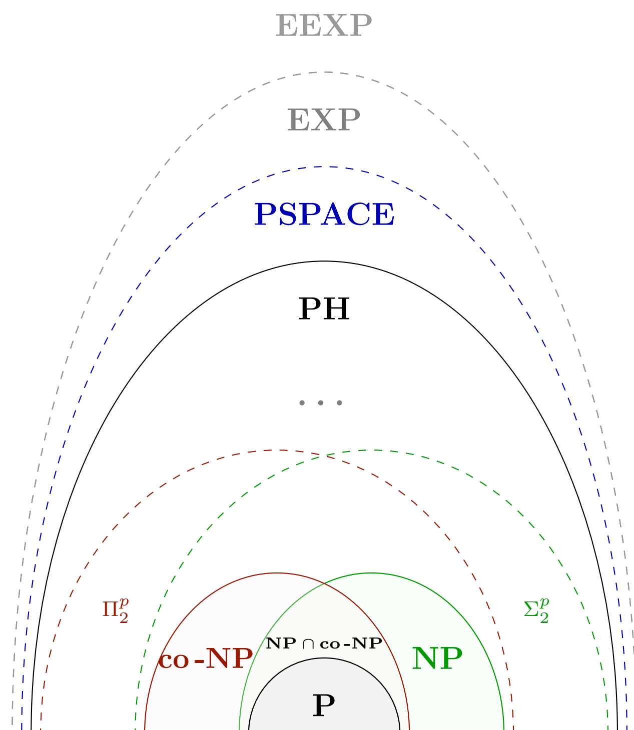


Figure 4: Classes below and above **PSPACE**.

Time vs Space. Why is it shown on that figure that **PSPACE** $\subseteq$ **EXP**? This is because for a space-constructible function $s(n) \geqslant \log n$,

$$\mathbf{DSpace}[s(n)] \subseteq \mathbf{DTime}[const^{s(n)}],$$

(this is due to the limit on the path length, i.e., the number of configurations; note that saying this we implicitly use the fact that we can compute the value of $s(n)$ and thus the value of $2^{s(n)}$).

Note that in the other direction

$$\mathbf{DTime}[t(n)] \subseteq \mathbf{DSpace}[t(n)]$$

is trivial: a machine cannot use more space than time. A striking very recent result by Ryan Williams is that we can even save some space: for time-constructible $t(n) \geqslant n$,

$$\mathbf{DTime}[t(n)] \subseteq \mathbf{DSpace}[\sqrt{t(n) \log n}] \text{ (Ryan Williams, 2025).}$$

PH vs PSPACE. We have shown that **PH** $\subseteq$ **PSPACE**. Can it be that they are equal? If **PH** $=$ **PSPACE**, then `QBF` belongs to **PH**, thus it belongs to Σ_k^p for some specific $k \in \mathbb{N}$. Thus every other problem in **PH** is reducible to a problem in Σ_k^p, but Σ_k^p is closed under polynomial-time many-one reductions, so every problem in **PH** belongs to this particular Σ_k^p; this would be a collapse of the polynomial hierarchy to its level k, which is unlikely!

13 Randomized computation

In the previous courses you have seen several examples of randomized algorithms, that is, algorithms that can sample random bits and use them; their answer might depend on the luck (on the actual values of these random bits), in particular, the answer may be wrong ("yes" instead of "no", and vice versa).

A randomized algorithm can be formalized in several ways.

Important points before we start:

- In this course we will talk only about randomized algorithms for decision problems, i.e., languages (not approximation algorithms! the answer is exact... but may be wrong).
- A randomized algorithm has access to a source of uniformly random independent bits (aka "random string").
- Such an algorithm can give a wrong answer with a not-so-big probability.
- There are several versions of randomized algorithms (to be defined later) w.r.t. the types of errors they are allowed to make:
  - "False negative": the correct answer is Yes, but we say No. (We will define the class **RP** for polynomial-time algorithms of this type.)
  - "False positive": the correct answer is No, but we say Yes. (We will define **co-RP**.)
  - Both false negative and false positive answers. (We will define **BPP**.)
  - **Zero** error. (We will define **ZPP**.) Yes, no error a all, so what is the use of randomness? There is a spoiler in the footnote[13].

13.1 One-sided bounded error

Definition 13.1. A one-sided bounded error algorithm for a language L is a probabilistic Turing machine A such that

- $\forall x \in L \qquad \Pr\{A(x) = 1\} \geqslant \frac{1}{2}$.   // *False negative with probability* $\leqslant 1/2$
- $\forall x \notin L \qquad A(x) = 0$.   // *No false positives*

The class $\mathbf{RTime}[t(n)]$ consists of the languages possessing such $O(t(n))$-time algorithms. Note that the running time bound $t(n)$ here does not depend on the randomness, it is guaranteed that the algorithm stops in time $t(n) \cdot const$.

$$\mathbf{RP} = \bigcup_{k \in \mathbb{N}} \mathbf{RTime}[n^k]$$

[13] The running time may depend on the randomness.

Example 13.1 (Fingerprinting: Verifying matrix multiplication)**.** One can multiply two $n \times n$ matrices using $O(n^3)$ (obviously) or slightly faster [Strassen], or even just $O(n^{2.3715...})$ arithmetic operations[14] [Virginia Vassilevska-Williams et al, 2023] — still even this new algorithm is much slower than n^2.

Can one *verify* it faster? We are aiming at $O(n^2)$ operations.

Randomized (**co-RTime**) algorithm for verifying the product [Freivalds, 1977]:
Input: $n \times n$ matrices A, B, C.
Output: "Yes" if $A \cdot B = C$.

1. Sample a random vector $r \in \{0, 1\}^n$.
2. Compute vectors $e_1 = A(Br)$ and $e_2 = Cr$. // *$O(n^2)$ operations*
3. If $e_1 = e_2$, then accept; otherwise reject.

It takes $O(n^2)$ operations, because this algorithm does not multiply matrices: it multiplies a matrix by a vector only ($s := Br$, then $e_1 := As$; also $e_2 := Cr$).

This is a **co-RTime** algorithm. Let us prove it:

- If $A \cdot B = C$, it accepts.
- If $A \cdot B \neq C$, let $D := AB - C$. This is a nonzero matrix; let us denote (some) its nonzero entry $D_{p,s} \neq 0$. How a false positive can occur? The matrix is nonzero, but the vectors e_1 and e_2 are equal.

$$\Pr\{\text{false positive}\} = \Pr_r\{A(Br) = Cr\} = \Pr_r\{Dr = 0\} = \Pr_r\left\{\sum_{i=1}^{n} D_{p,i} r_i = 0\right\}$$
$$= \Pr_r\Big\{ -\sum_{i \neq s} D_{p,i} r_i = D_{p,s} r_s \Big\} \leqslant \frac{1}{2}.$$

The probability is at most $\frac{1}{2}$, because r_i's are independent, so we can think like we choose r_s after $\sum_{i \neq s} D_{p,i} r_i$ is computed). Thus bad luck means that the value of r_s is accidentally chosen to be $D_{p,s}^{-1} \sum_{i \neq s} D_{p,i} r_i$. However, we choose it at random from two possibilities (0 and 1)!

Probability amplification: one-sided version. The probability of error in Def. 13.1 is a huge error probability, can one do better? Yes!

Success probability amplification (that is, error reduction) can be made by repeating the algorithm.

Let A be a **RTime** algorithm for L with error probability $1/2$. Consider the following algorithm that decides the same language:

[14] Operations on the scalars that are written in the entries of these matrices, like integer $+$ and $\cdot$ if these are integer matrices.

```
Amplified algorithm B.
Input:  x

For i := 1 to k
-- if A(x) = 1 then accept.            // All attempts use independent random bits!
Reject.
```

Nothing has changed if $x \notin L$, the algorithm still rejects it. However, note that if $x \in L$, then $\Pr\{B(x) = 0\} \leqslant \frac{1}{2^k}$.

Therefore, if A runs in polynomial time and k is a constant or even a polynomial, the running time of B is still polynomial! So an equivalent definition of **RP** could use another constant strictly between 0 and 1 — not necessarily $\frac{1}{2}$. The definition of **RP** is robust w.r.t. such changes. The definition of **RTime** is not: we remain in the same class **RTime**$[t(n)]$ as long as we repeat it a constant number of times, but at least it allows us to change the constant in Def. 13.1!

Even smaller success probability can be amplified. If we have an algorithm that has an error $1 - \dfrac{1}{T(n)}$ (where T could be a function of the input size, and even something exponentially depending on it!), then repeating it $T(n)$ times would turn it into a one-sided bounded-error algorithm according to **RTime** definition above:

$$\left(1 - \frac{1}{T(n)}\right)^{T(n)} < \frac{1}{\mathrm{e}}.$$

It means that informally once we have a $q(n)$-time algorithm with probability of success at least $\frac{1}{T(n)}$, we can turn it into a bounded-error algorithm that will run in time $O(T(n)q(n))$.

Example 13.2 (An important example: Polynomial Identity Testing)**.** Before we formulate this problem, let us start with a related one, `ZeroP`:

Consider an "algebraic" circuit C that instead of Boolean operations is using arithmetic operations $(+, \cdot)$, variables x_i, constants $0, 1, -1$. Such a circuit computes a polynomial in $p_C \in \mathbb{Z}[x_1, \ldots, x_n]$. (Every node of this circuit computes some polynomial: the input x_1 computes the polynomial x_1, the product of two inputs x_1 and x_2 computes the polynomial $x_1 x_2$, etc. — see Fig. 5 for example.)

Question: Is it a **zero polynomial**? (That is, are its coefficients zeroes?)

(Think of $x_1, x_2, \ldots$ as integer variables or variables from (for example) a finite field. One can compute the result of the circuit as a number. However, the question is not about this value: it is about the coefficients of the polynomial; of course, if they are zeroes, then the value is also zero — but not necessarily vice versa.)

Now the following problem known as `PIT` (Polynomial Identity Testing) is easily reducible to `ZeroP`:

Question: Consider two algebraic circuits C_1, C_2. Do they compute the same polynomial? (That is, is each coefficient in C_1 equal to the coefficient for the same monomial in C_2?)

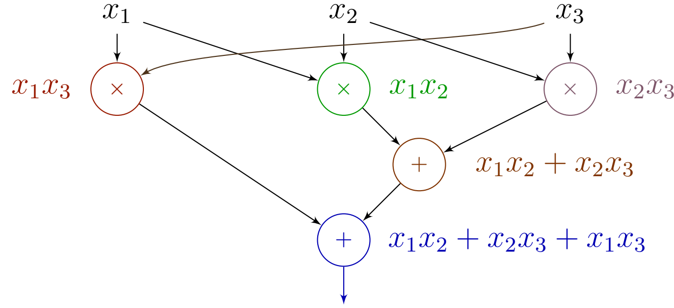


Figure 5: An algebraic circuit (example). Polynomials actually computed by internal nodes are shown near these nodes.

Reduction: Consider the circuit $C_1 \cdot (C_2 \cdot (-1))$. It computes the zero polynomial if and only if C_1 and C_2 compute the same polynomial.

Now let us come back to `ZeroP`.

```
Randomized algorithm for ZeroP in co-RP.
Input:  C.

Select random values r_1, ..., r_n for x_1, ..., x_n.
If C(r_1, ..., r_n) = 0, then answer ‘‘Yes’’, otherwise answer ‘‘No’’.
```

This algorithm exhibits no false negatives.

To estimate the probability of error, we need to specify from what set $r_1, \ldots, r_n$ are taken. In fact, it depends on the degree of our polynomial.

Let $\deg p$ denote the (full) degree of a polynomial p (that is, the maximum degree of a monomial in p, where the degree of a monomial is the sum of degrees of each variable occurring in it — for example, the degree of $10x_1^2x_2$ is 3).

The following lemma will help us:

Lemma 13.1 (Schwartz–Zippel)**.** *Let $\mathbb{F}$ be a field or $\mathbb{F} = \mathbb{Z}$, $0 \neq p \in \mathbb{F}[x_1, \ldots, x_n]$, $\deg p \leqslant d$. Let finite $S \subseteq \mathbb{F}$. Then*

$$\Pr[p(r_1, \ldots, r_n) = 0] \leqslant \frac{d}{|S|},$$

where r_i's are chosen uniformly and independently from S.

By this lemma if we choose the random values $1 \leqslant r_i \leqslant s \cdot \deg p_C$, then the probability of error $\leqslant 1/s$.

The only problem is that we do not know $\deg p_C$. If for some reason it is polynomial, we can select r_i's from an interval of polynomial length. While we will be computing $C(r_1, \ldots, r_n)$, the intermediate values will still have a polynomial number of bits.

This is also useful. In some applications we know that the degree of the polynomial is at most polynomial, or even less than that.

If we do not have a good estimate of $\deg p_C$, the only thing we only know is $\deg p_C \leqslant 2^{|C|}$. (Yes, it can be that large! Think about the circuit that repeatedly squares a variable: $g_1 = x_1 \cdot x_1$, $g_2 = g_1 \cdot g_1$, ..., $g_n = g_{n-1} \cdot g_{n-1}$.)

If we choose $r_i \leqslant 2^{|C|}$, it's OK, but when we will be computing $C(r_1, \ldots, r_n)$, the intermediate values could be doubly exponential (because of the large degree: even when $r_i = 2$, the exponential degree of it produces a doubly exponential number).

There are two possibilities to overcome this problem.

The first one is to choose a large finite field from the beginning and work in it (this approach would work even if instead of polynomials with integer coefficients we consider polynomials with $GF(2)$ coefficients: we can use a large enough extension field of $GF(2)$).

The second one (that works at least for $\mathbb{F} = \mathbb{Z}$) is simply to choose a large prime $m \in \mathbb{P}$ and work modulo that prime. Then we will have three sources of error:

1. The calculations that we would make before (without modulo arithmetic) would already lead to an error — we hit a root in the Schwartz–Zippel lemma; this probability is given by the lemma;

2. These calculations would be OK (that is, $p_C(r_1, \ldots, r_n) \neq 0$), but modulo m they are not, that is, $p_C(r_1, \ldots, r_n) \equiv 0 \pmod{m}$. It means that m is a divisor of this number $\rho := p_C(r_1, \ldots, r_n)$. A k-bit number cannot have more than k distinct prime divisors. Our number can be large, namely, it is $\leqslant K^{2^{|C|}}$, where K is the largest number in the interval we choose; however, the number of bits in it is just exponential in $|C|$. If m selected at random from a set of $2^{2|C|}$ primes, it would be enough: by the Prime Number Theorem there are $\sim N/\ln N$ primes less than N, thus one can take $K \sim 2^{4|C|}$, we will have enough primes, and the number of bits (and thus prime divisors) in our ρ is $O(2^{|C|}4|C|)$, much less than $2^{2|C|}$.

3. We need to select this large prime. We can do it by sampling a number repeatedly at random and verifying it, say, using the known deterministic polynomial-time algorithm for primality. The chances that we succeed in one trial are $\sim 1/\ln 2^{4|C|} = \Omega(1/|C|)$, thus after $O(|C|)$ trials we will get any particular constant probability of error we wish (the error here is that we did not succeed in finding a prime number and thus we have to accept blindly and possibly err).

13.2 Two-sided bounded error

Definition 13.2. A two-sided bounded error randomized algorithm for a language L is a probabilistic Turing machine A such that

- $\forall x \in L \qquad \Pr\{A(x) = 1\} \geqslant \dfrac{3}{4}.$ // *False negative with probability* $\leqslant 1/4$
- $\forall x \notin L \qquad \Pr\{A(x) = 1\} \leqslant \dfrac{1}{4}.$ // *False positive with probability* $\leqslant 1/4$

The class **BPTime**$[t(n)]$ consists of the languages with such $O(t(n))$-time algorithms.

$$\mathbf{BPP} = \bigcup_{k\in\mathbb{N}} \mathbf{BPTime}[n^k]$$

Success amplification in BPP. We cannot reduce the error probability like we did for **RTime**, but we can do it almost as efficiently.

Let A be a **BPTime** algorithm for L with error probability $1/4$. Consider the following algorithm that takes the majority vote:

```
Amplified algorithm B.
Input:  x
For i := 1 to k
-- if A(x) = 1 then a_i := 1 else a_i := 0.//all attempts use independent random bits!
If ∑_{j=1}^k a_i > k/2 then accept else reject.
```

Then $\Pr\{\text{more than } k/2 \text{ errors}\} \leqslant 2^{-\Omega(k)}$. Why?

Recall one of Chernoff's inequalities:

$$\Pr\{Z > (1+\varepsilon)pk\} < \left(\frac{e^{\varepsilon}}{(1+\varepsilon)^{1+\varepsilon}}\right)^{pk} \leq e^{-\frac{pk\varepsilon^2}{4}},$$

where $Z = \sum_{i=1}^{k} z_i$ and every z_i is an independent random variable that equals 1 with probability p and 0 with probability $1-p$.

For us z_i is an error ($a_i \neq L(x)$), in the i-th attempt, $p = \frac{1}{4}$, $\varepsilon = 1$.

13.3 Another view of RP and BPP

If we have a $p(n)$-time bounded-error polynomial-time randomized algorithm, we can view it in different equivalent ways. In all the cases we can assume that it uses exactly the same amount of random bits for the same input length (for example, just $p(n)$ unless we are interested in lowering their number).

A probabilistic Turing machine can be viewed as

- An algorithm that tosses a (fair) random coin, or
  a DTM that has a dedicated readonly random tape (this is the default view).

  This approach allows to use the probability theory: nicer proofs if you remember it!

- A witness-type NTM where witness is the random tape.

  This is the same, but it is useful if we count the witnesses in an easy combinatorial way.

- A NTM making decisions – just these decisions are now randomized, not nondeterministic.

  Then the run of such an algorithm can be visualized as a computation tree, but contrary to nondeterministic computations we think about the share of accepting/rejecting paths, not merely their exisence.

For example, let us redefine **RP** and **BPP** using a witness-type NTM:

$L \in$ **RP** if there is a polynomially bounded, polynomial-time verifiable relation $R \subseteq \{0,1\}^n \times \{0,1\}^{p(n)}$ such that $\forall x \in \{0,1\}^*$

$$\begin{aligned} x \notin L &\Rightarrow \forall w\ (x,w) \notin R \\ x \in L &\Rightarrow \frac{|\{w \mid (x,w) \in R\}|}{|\{\text{all } w\}|} \geqslant \frac{1}{2}. \end{aligned}$$

$L \in$ **BPP**, if there is a polynomially bounded, polynomial-time verifiable relation $R \subseteq \{0,1\}^n \times \{0,1\}^{p(n)}$ such that $\forall x \in \{0,1\}^*$

$$\begin{aligned} x \notin L &\Rightarrow \frac{|\{w \mid (x,w) \in R\}|}{|\{\text{all } w\}|} \leqslant \frac{1}{4}, \\ x \in L &\Rightarrow \frac{|\{w \mid (x,w) \in R\}|}{|\{\text{all } w\}|} \geqslant \frac{3}{4}. \end{aligned}$$

Thus we can think about **RP** and **BPP** computations in terms of computation trees.

13.4 Errorless randomized algorithms

Languages decided by errorless polynomial-time randomized algorithms can be defined in two equivalent ways:

Definition 1. **ZPP** = **RP** $\cap$ **co-RP**.

Definition 2. $L \in$ **ZPP** if there is an errorless expected[15]-polynomial-time algorithm for L.

Lemma 13.2. *These two definitions of* **ZPP** *are equivalent.*

Proof. $\boxed{2 \Rightarrow 1}$ Let A have $\mathbb{E}\,\mathrm{time}_A(x) \leqslant p(|x|)$.

Interrupt A after $2p(|x|)$ steps and answer "no" (false negative).

$\Pr\{\text{time} \geqslant 2 \cdot \mathbb{E}\,\text{time}\} \leqslant \frac{1}{2}$ by Markov's inequality.

Thus we get an **RP** algorithm.

Analogously we get a **co-RP**-algorithm (if we interrupt the errorless algorithm, we answer "yes" and are punished by false positives.

$\boxed{1 \Rightarrow 2}$ Let B be an **RP**- and C be a **co-RP**- algorithm for L.

[15] That is, there is a polynomial $p(n)$ such that for every input x of length n the mathematical expectation of the running time on x is bounded by $p(n)$.

Note that the answer "yes" for an **RP** algorithm is always correct, and the answer "no" for a **co-RP** algorithm is always correct. There is some specific correct answer for our input $x \in L$, we just don't know which one.

Thus we run $B(x)$ and $C(x)$ one by one again and again until one of them produces a provably correct answer for x (one of them is capable of doing that). How much time will it take?

If the error probability is $\frac{1}{2}$,

$$\mathbf{E}\, attempts \leqslant \frac{1}{2} \cdot 1 + \frac{1}{4} \cdot 2 + \frac{1}{8} \cdot 3 + \ldots = \sum_{k=1}^{\infty} \frac{k}{2^k} = 2.$$

If the error probability is better, the running time will be, of course, smaller (if you are not convinced by that, assume that the error probability for this particular x is $p \leqslant \frac{1}{2}$ and compute the expectation of time yourself in terms of p as an exercise). □

Example 13.3 (Errorless algorithm example: pattern matching with verification)**.** Consider the following problem:

FIND A PATTERN IN A STRING.
Input: $p, s \in \{0,1\}^*$.
Output: "yes" if $\exists x, y$ s.t. $s = xpy$.

- This problem is in **NP** by definition.
- A brute-force deterministic algorithm makes $\leqslant |s||p|$ comparisons.
- In a previous course we learned a deterministic $O(|s|)$ time algorithm: The Knuth–Morrison–Pratt algorithm, which is rather complicated.
- As an example will show now a simple randomized $O(|s|)$-time algorithm.

Algorithm:

1. Choose a random $m \in \mathbb{P}$ among at least M (to be chosen) first primes:
   Due to the Prime Number Theorem there are $\sim N/\ln N$ primes less than N (there is a specific constant in this asymptotic bound so that we know that there are at least as many primes less than a given natural number).
2. We identify $|p|$-bit strings with natural numbers $0\,.\,.\,2^{|p|} - 1$.
3. Compute $r := p \bmod m$.
4. For i:=1 to $|s| - |p| + 1$
   (a) Compute $r' := s[i..i + |p| - 1] \bmod m$. // *one can recompute r' faster* // *from its previous value:* // *we drop one bit on the right and add one bit on the left*

(b) If $r = r'$ (as numbers $< m$) then
if $s[i..i + |p| - 1] = p$ (as strings) then // *safety check*
accept.

5. Reject.

If we won't do the safety check, we would get a **co-RTime** algorithm:

- If pattern p occurs in s, the algorithm accepts.
- If pattern p does not occur, there may be false positive with probability $\leqslant \frac{(|s|-|p|+1)|p|}{M}$ (because a k-bit number has at most k prime divisors, so at each position i there are at most $|p|$ divisors of the number $p - s[i..i + |p| - 1]$, $(|s| - |p| + 1)|p|$ prime divisors in total).
- Choose $M \sim 2|p||s|$, thus the error probability is at most $1/2$.

Now as we do the safety check, the algorithm becomes zero-error; its expected time is polynomial:
$\mathbb{E}$ operations $\leqslant c|s| + O(|p|) + \frac{|s||p|}{m} \cdot |s| \cdot |p| = c|s| + O(1) + O(|p|) = c|s| + O(|p|)$
if $M \sim |s|^2|p|^2$, where c is a small constant (the number of operations to recalculate r') and $O(|p|)$ is for the initialization.

Remark 13.1. Note that we compute the number of operations. To estimate the running time on, say, a RAM machine according to the log-cost criterion one needs to take into account the cost of recalculating the hash modulo m as well as the size of the addresses of elements of the arrays, but for the KMP algorithm one needs to do the same for fair comparison.

There is no point in diving into these details, however, — this example is an illustration of how we can build (sometimes) a zero-error algorithm starting with a bounded-error algorithm.

13.5 Observations: Where randomized algorithms are on the map?

Trivially **RP** $\subseteq$ **NP** (in **NP** we have at least one correct witness for $x \in L$; in **RP** we must have at least have of them correct).

Thus **ZPP** $=$ **RP** $\cap$ **co-RP** $\subseteq$ **NP** $\cap$ **co-NP**.

Where is **BPP**? It is unknown whether **BPP** $\subseteq$ **NP** or **NP** $\subseteq$ **BPP**. Yet **BPP** $\subseteq \Sigma_2^p \cap \Pi_2^p$ (we won't prove it in this course).

In particular, **BPP** $\subseteq$ **PSPACE** $\subseteq$ **EXP**. To see this without showing the above unproven fact, assume w.l.o.g. that a random string has a fixed polynomial length $p(n)$. Then to simulate deterministically a **BPP** algorithm A for a language L, on input x of length n enumerate all random strings of string $p(n)$ and run A on x with each random

string — count the number (the share) of the answers "yes", thus get the probability of acceptance and the result $x \in^? L$.

The two symmetric classes of randomized algorithms that we studied (**ZPP** and **BPP**) are closed under using subprocedures of the same type.

- Recall that $\mathbf{P} = \mathbf{P}^{\mathbf{P}}$ (we just compute the oracle's answer ourselves).
- $\mathbf{BPP} = \mathbf{BPP}^{\mathbf{BPP}}$ (if the error probability of the oracle is exponentially small — we learned how to do that — then with an overwhelming probability all its (polynomially many!) answers will be correct, so the probability of error of the overall algorithm will be close to that of the lower machine).
- $\mathbf{ZPP} = \mathbf{ZPP}^{\mathbf{ZPP}}$ (left as an exercise — you can use the definition with expected polynomial time to solve it).
- $\mathbf{RP}^{\mathbf{RP}} = ?\ldots$ unknown; this way one can indeed build a hierarchy inside **BPP**! However, since many people believe even that $\mathbf{P} = \mathbf{BPP}$, it is not a very popular hierarchy.

Remark 13.2. A word of caution. Consider what happens, say, if $\mathbf{NP} \subseteq \mathbf{BPP}$. One could try to show more under that assumption: $\mathbf{NP}^{\mathbf{NP}} \subseteq \mathbf{NP}^{\mathbf{BPP}} \subseteq \ldots$ however, here one should stop and think between replacing the lower **NP** by **BPP**. Recall that a language belongs to $\mathbf{NP}^{\mathbf{BPP}}$ iff it can be decided by a nondeterministic oracle polynomial-time Turing machine with an oracle in **BPP**. That is, the notation "**NP**" for the lower class here (actually, $\mathbf{NP}^{\bullet}$, but $\bullet$ is replaced by an oracle [class]) stands for machines (in particular, oracle machines), not languages. However, our assumption $\mathbf{NP} \subseteq \mathbf{BPP}$ says nothing about oracle machines. If one could tell us that $\forall O$, $\mathbf{NP}^{O} \subseteq \mathbf{BPP}^{O}$ (in such a case they say that this inclusion is relativizable), we could proceed to $\mathbf{NP}^{\mathbf{BPP}} \subseteq \mathbf{BPP}^{\mathbf{BPP}} = \mathbf{BPP}$. However, without that we need other methods to prove it (*exercise* — this is possible, consider a witness-like definition of **NP**, define new languages, use **BPP** error reduction, etc, this is not straightforward but doable).

References

[Sip] M. Sipser. Introduction to the Theory of Computation.

[AB] S. Arora, B. Barak. Computational Complexity: A Modern Approach.