Computational Models. Part II
DRAFT Lecture Notes

Edward A. Hirsch*

Abstract

These are lecture notes for the second part of the course that have been taught in Spring 2025. This part mostly concerns Turing machines. These lecture notes assume familiarity with the first part (necessary references are given in the Prerequisites section), though they can be read essentially independently.

This course is a prerequisite for the Computability course.

*Ariel University

Contents

1 Pre-requisites **3**

2 Decision and search problems **4**

3 Automata with different types of memory, the Church–Turing Thesis, and its limits **5**
3.1 Memory models . . . 5
3.2 Purely mechanical logical computing machines . . . 6
3.3 Uniform and non-uniform models . . . 6

4 Deterministic Turing machines **9**
4.1 Definitions . . . 9
4.2 The binary alphabet . . . 13
4.3 The number of tapes . . . 16
4.4 A time lower bound for 1-tape DTMs . . . 19
4.5 A note of data representation (encoding) . . . 22
4.6 Encodings of Turing machines. Turing numbers . . . 22
4.7 A Universal Turing machine . . . 23
4.8 RAM: Random Access Memory and Random Access Machine . . . 25
4.8.1 Definition: A specific instruction set . . . 25
4.8.2 The running time of a RAM program . . . 27
4.8.3 Equivalence to DTMs . . . 28

5 Nondeterministic Turing machines **29**
5.1 Nondeterministic Turing machines: Definition . . . 29
5.2 Computation trees . . . 30
5.3 The running time and the result of a NTM . . . 30

6 Decision and recognition **31**
6.1 Decision and recognition, recursive and recursively enumerable languages . . . 31
6.2 DTMs and NTMs are equivalent wrt recursive enumerability . . . 33
6.3 Enumerators . . . 33

7 Classes of languages (decision problems) **35**
7.1 Classes for unbounded-time computation: **R** and **RE** . . . 35
7.2 Classes for bounded-time computation . . . 38
7.2.1 Classes **DTime** and **P** . . . 38
7.2.2 Classes **NTime** and **NP** . . . 39
7.3 The complement . . . 43
7.4 General picture and major open questions . . . 43

8 P.S. Search problems **44**
8.1 **NP**, version 3 . . . 45
8.2 A version of **NP** for search problems . . . 46
8.3 Levin's optimal algorithm for **Search-NP** problems . . . 48

1 Pre-requisites

It is strongly recommended to start reading these notes after you have made yourself familiar with the notions of

- formal languages (and decision problems associated with them),
- finite automata (DFA and NFA).

In this section we remind some basic notation (see [Sipser, Chapter 1] for more details). These lecture notes also mention (just a little bit) context-free grammars — they are defined, for example, in [Sipser, Section 2.1].

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 most important operation defined over strings is concatenation: if we are given two strings $a = a_1 a_2 \dots a_k$ and $b = b_1 b_2 \dots b_m$, where $a_1 = a[1], \dots, a_k = a[k]$ and $b_1 = b[1], \dots, b_m = b[m]$ are letters, then the contatenation of a and b is the string $a \circ b = a_1 a_2 \dots a_k b_1 b_2 \dots b_m$. Sometimes we omit $\circ$. Some sources use $\cdot$ instead of $\circ$. We denote the empty string by ε. We denotes the length of a string a by $|a|$ (for example, $|110| = 3$).

Since languages are sets, one can use set-theoretic operations over them. In addition, one can use concatenation

$$A \circ B = \{ab \,:\, a \in A,\ b \in B\},$$

define degrees

$$A^k = \underbrace{A \circ \dots \circ A}_{k \text{ copies}},$$

and Kleene star:

$$A^* = \bigcup_{i \geqslant 0} A^i.$$

Note that $\varepsilon \in A^*$.

When we are talking about a language, it is important to fix the alphabet that we are working with — for example, to compute correctly the complement $\overline{L}$ of a language L: if we are working in the alphabet Σ, it means that $L \subseteq \Sigma^*$ and $\overline{L} = \Sigma^* \setminus L$.

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").

2 Decision and search problems

To solve a computational problem, one must design a device that can respond to certain questions (inputs) — and there may be infinitely many inputs.

Decision problems assume one of the answers "yes" or "no". Several examples:

- Is this string s of the form $a^n b^n$ for some $n \in \mathbb{N}$?
  (This is something we frequently talked about during the first part of the course.)
- Is this number n prime (or composite)?
- Does this graph G contain a cycle?
- Will this program p ever stop on input x?
  (This is an incredibly important type of questions!)

We already know from the first part of the course that a decision problem is associated with a language L, that is, it can be reformulated as "on input x, decide whether $x \in L$"?

To be absolutely formal, we say that $L \subseteq \Sigma^*$ and $x \in \Sigma^*$, where Σ is a finite alphabet. Certainly all the examples above can be encoded in a finite alphabet (and even in the binary alphabet $\{0, 1\}$).

Search problems ask to find a solution (a certain string satisfying certain constraints). Several examples:

- Given a number n, find any nontrivial divisor p (that is, $p \in \mathbb{N}$, $p|n$, and $p \neq 1, n$).
- Given a program p, find any input x such that p stops on it.
- Given a graph G, find a cycle in it.
- Given x, compute $f(x)$. (Here $f : \{0, 1\}^* \to \{0, 1\}^*$ is a function.)

Of course, there may be more than one solution, thus different (correct) algorithms for a search problem may output different solutions.

To be absolutely formal, we say that a search problem is associated with a relation $R \subseteq \Sigma^* \times \Sigma^*$, and in order to solve this problem, given a "question" $x \in \Sigma^*$ on the input, one must output a string $w \in \Sigma^*$ (called a "solution") such that $R(x, w)$ is true. It may happen that there is no such w — then output "there are no solutions".

A particular case of a search problem is computing a function. In this case there is always exactly one solution.

3 Automata with different types of memory, the Church–Turing Thesis, and its limits

3.1 Memory models

In the first part of the course we studied thoroughly one computational model: the finite automaton. We found that deterministic (DFA) and non-deterministic (NFA) versions of this device have the same computational power (that is, they accept the same languages) though the number of states may increase a lot if we convert an NFA into a DFA.

We have seen that there are very simple languages that finite automata cannot accept (such as the language containing all the palindromes: the words that do not change if one reads them left-to-right or right-to-left). This is due to the fact that a finite automaton has no memory — or, more exactly, its memory is limited to the number of its states, that is, $O(1)$ bits. In other words, the size of its memory (space) does not depend on the size of the input.

There are computational models *(which we do not study in detail in this course)* that do use unbounded memory but still do not reach the full power of what we think as algorithms (computers, computer programs). For example, a pushdown automaton uses a stack of unlimited size (also called magazine to emphasize that only its top is easily reachable). A stack is a one-sided sequence of cells with two operations: `push` (add a cell on the top and write a new letter into it) and `pop` (read the letter from the cell on the top and throw out this cell thus clearing the access to the next cell). Nondeterministic pushdown automata accept exactly the same languages as context-free grammars can generate (we did not prove it in the course, while this is not a very difficult exercise: to "execute" the right side of a rule, push the symbols into the stack and pop a symbol back when we are ready to process it). However, still there are very simple languages that such automata cannot accept — because their mode of access to the memory requires them to throw out all the information that lies on top of the information that is needed at the moment.

Example 3.1. *An animated example of a nondeterministic pushdown automaton for* $\{xx^R : x \in \Sigma^*\}$ *is given in the slides for Lecture 5. It is easy to transform it into an automaton accepting the language of all palindromes (includes those with an odd length).*

General-purpose computational models are those that express the same power as algorithms (or computers, or programs written using programming languages that can access unbounded memory — at least through reading and writing files of unbounded size). We will see next that the main features for such a model are

- A read-write memory of potentially unbounded size (that is, the required size may depend on the input),
- This memory can be inefficient, but still must allow to access every memory cell without forgetting the contents of other memory cells (except for a small number of temporary registers, if they are needed),
- A potentially unbounded running time (the number of steps, where at each step a single instruction of a "program" is executed).

3.2 Purely mechanical logical computing machines

In the first half of the XX century, several mathematicians came up with a definition of an algorithm and of [algorithmically] decidable problems. They called this notion differently, but the models turned out to be equivalent: Alan Turing's machines, Emile Post's combinatorial processes, Kurt Goedel's and Stephen Kleene's "recursive functions" (in particular, Goedel proved his famous Goedel's Theorems on unprovability essentially talking about (un)decidability), Alonzo Church's λ-calculus (which is the base of Functional Programming today).

Church–Turing Thesis is an informal statement saying that deterministic Turing machines (or, equivalently, one of the other equivalent models) can compute everything that we can compute "mechanically" (that is, step by step, in a finite amount of time, using finite but unbounded resources, by rigorously defined laws — called a program — that tell us what to do at the next step without using our human intellect).

Note that this thesis says nothing about the running time (it can vary for different models of computation) and says nothing about intricate tasks like communication protocols implemented using analog (for example, quantum) physical devices. It only says what can be computed in a finite amount of steps on a mechanical device with a single program (not depending on the input or its size).

In fact, randomized computations and quantum computations (with a bounded probability of error) still satisfy this thesis though may run much faster (sometimes). Just to repeat it — we are not talking about quantum communication here.

Remark 3.1. *Unbounded and unrestricted memory space can be made according to different architectures, such as one tape (or file), several tapes, or RAM (random access memory). In what follows we will see that there is no much difference between them.*

3.3 Uniform and non-uniform models

There are computational problems that cannot be solved using deterministic Turing machines (or modern computers) in any amount of time. Are there stronger computational models that can solve (some of) them? Yes!

First of all, there are models that are less predictible than algorithms: for example, they can behave non-deterministically. We will take about such models later, but there is something even more spectacular: non-uniform models.

We call a model uniform if it can be described by $O(1)$ bits, that is, there is a single program that one can apply to inputs of any size. DFAs and NFAs are certainly uniform models: to accept a regular language, one needs a single automaton that can read words of any length. Turing machines (deterministic, and even non-deterministic) are also a uniform model.

On the contrary, a non-uniform model allows one to use infinitely many devices for accepting a language (or performing another computational task), or to use one device described by infinitely many bits. A specific device in this collection (or a piece of an infinite device) can be applied, for example, to inputs of a certain size.

Families of Boolean circuits are an important example of a non-uniform computational model.

Definition 3.1. *A Boolean circuit C is a directed acyclic graph. Its nodes are called gates.*

- *Its sources (vertices of in-degree zero) correspond to the input bits (called also "variables" or even just "inputs") $x_1, x_2, \ldots, x_n$.*
- *Each internal node is labeled by a unary or a binary Boolean function. Its in-degree is 1 or 2, respectively, and the incoming edges come from the arguments of the function. Its out-degree may be arbitrary: the result of the function goes through the outgoing edges (informally, it means that the result of the operation, once computed, can be used many times).*
- *Some of the gates are labelled as output bits $o_1, o_2, \ldots, o_m$. (One can think of them as vertices of out-degree 0 though this is not necessary.)*

A circuit with n input bits and m output bits computes a function $\{0,1\}^n \to \{0,1\}^m$ in a natural way: its input $x \in \{0,1\}^n$ is split into the bits $x_1, x_2, \ldots, x_n$ and is fed to the sources of the graph. Then the value of a gate that has already received the values for all its arguments through the incoming edges is determined, thus we can evaluate more gates. So the values of the next such gates are determined, etc, until we evaluate all the output gates. We write $C(x)$ to denote the string of 0/1-values that will eventually appear in the output gates of C.

An important case of a Boolean circuit is a single-output circuit: it computes a predicate, that is, says either 1 ("yes") or 0 ("no").

Example 3.2 (Majority of 3 bits)**.** *The majority function outputs the majority of its input bits:*

$$\mathsf{Maj}_n(x_1, x_2, \ldots, x_n) = \begin{cases} 1, & \text{if } \sum_{i=1}^{n} x_i > \frac{n}{2} \\ 0, & \text{otherwise} \end{cases}$$

The circuit given in Fig. 1 computes Maj_3, that is, it outputs 1 if at least two of its three inputs bits are true. In other words, it accepts the words 111, 110, 011, and 101, and does not accept 000, 001, 010, 100.

Slides for Lecture 5 contain more examples of Boolean circuits.

Obviously, a circuit with n input gates can be applied to inputs of size n only. Thus in order to compute a function or check whether a word belongs to a language, for inputs of arbitrary length, one needs infinitely many circuits.

Definition 3.2. *A family of Boolean circuits is an infinite sequence $\{C_0, C_1, C_2, \ldots\}$ of Boolean circuits, where the circuit C_i has i input bits.*

One can think of functions or languages computed or decided by families of Boolean circuits. This is a non-uniform model, and it can accept many more languages than uniform models. (Recall an argument that if our devices can be described as finite strings, then they form a set

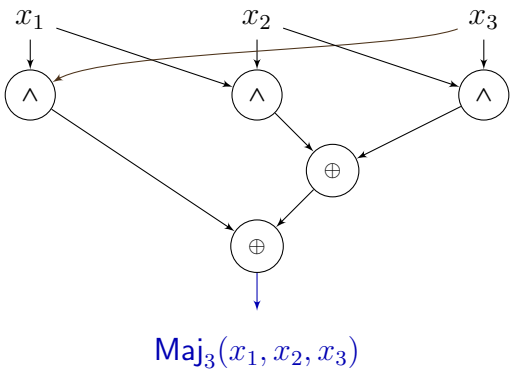


Figure 1: A Boolean circuit for Maj_3. Note that the two $\oplus$-gates can be replaced by $\vee$-gates.

of cardinality at most $\aleph_0$, while there are $2^{\aleph_0}$ possible languages even in the unary alphabet.) In the future, we will prove it more rigorously.

Of course, we cannot build a computer of infinite size. However, this model of computation is still very important:

- In an ordinary computer, operations are implemented as hardwired circuits of certain size (for example, for adding 64-bit integers). It's hardware — once manufactured, its size is fixed! If we have data of a different size, then we need to use such circuits several time in order to process it (for example, to take a sum of 128-bit or n-bit integers) or even do something else.

- In cryptography, there is a custom to use cryptosystems that use keys of a certain fixed size (say, 128 or 1024) that seems safe for now. If an adversary constructs a circuit that breaks precisely keys of the size that we are using, it is too bad and it does not matter for us how this circuit was constructed: by an algorithm or by a genius.

- A uniform version (see below) of families of Boolean circuits is a standard model for studying the complexity of parallel algorithms.

If all circuits in a family can be printed algorithmically, we fall back again to a uniform model of computation.

Definition 3.3. *A family of Boolean circuits $\{C_0, C_1, C_2, \ldots\}$ is called uniform if there is an algorithm (or, more formally, a deterministic Turing machine — defined below) that, given an integer i on input, of Boolean circuits, prints the circuit C_i (the one that has i input bits).*

Remark 3.2. *Usually it is assumed that i is given in unary, that is, as i ones: 1^i. It does not matter for us now. In complexity theory, they talk about various types of uniform families depending on the complexity of the generating algorithm (polynomial-time-uniform, logspace-uniform, etc).*

Remark 3.3. *Note that an optimal (the smallest possible) circuit for Maj_n can look very differently for different values of n. While it is easy to design a very inefficient algorithm that would print such an optimal circuit given n, it is unclear how to design an efficient algorithm for this task.*

What does it mean to print a circuit (if we are not talking about actually producing hardware)? One can think of a certain reasonable bit representation of this labelled directed graph: after all, everything can be represented as a bit string. We will talk about "reasonable representations" in more detail later in this course.

Exercise 3.1. *Generalize Example 3.2: Build a uniform family of Boolean circuits for the majority function of an arbitrary number of bits.*

Exercise 3.2 (You can attempt this exercise after you learn the definition of a Turing machine in the next section)**.** *It is not difficult to see that given a uniform family of circuits, one can build a single deterministic Turing machine that computes the same function.*

It is a bit more difficult (but still doable) to convert a deterministic Turing machine into an equivalent uniform family of circuits.

4 Deterministic Turing machines

4.1 Definitions

A deterministic Turing machine. A deterministic Turing machine performs deterministic computations using a tape (or several tapes) that is infinite in one side. A tape is a sequence of cells, each cell is capable of storing a single letter of a certain alphabet. A read/write head is associated with a tape. This head observes a single cell of a tape, and it can move left or right only one cell at a time, and it can read from or write to the cell it observes at the moment. If a machine has several tapes, their heads behave independently of each other.

Note that while a machine may have many tapes, it has a single control device (executing the machine's program), thus at each step it is in a single state.

Definition 4.1. *A k-tape deterministic Turing machine (k-DTM, DTM) is an ordered 9-tuple $(Q, \Sigma, \Gamma, \triangleright, \_, q_S, q_Y, q_N, \delta)$, where*

1. *Σ is the input alphabet,*
2. *$\triangleright \notin \Sigma$ is the start symbol,*
3. *$\_ \notin \Sigma$ is the blank symbol, $\_ \neq \triangleright$,*
4. *$\Gamma \supseteq \Sigma \cup \{\triangleright, \_\}$ is the tape alphabet,*
5. *Q is the set of states,*
6. *$q_S \in Q$ is the starting state,*
7. *$q_Y \in Q$ is the accepting state,*
8. *$q_N \in Q$ is the rejecting state, $q_N \neq q_Y$,*
9. *$\delta : Q \times \Gamma^k \to Q \times \Gamma^k \times \{\leftarrow, \rightarrow, \cdot\}^k$ is the program (transition function).*

Sometimes we denote $\rightarrow$ *by* $+1$, $\leftarrow$ *by* -1, *and* $\dot{-}$ *by* 0.

We consider only correctly formed Turing machines. (What follows is given in the notation for 1*-DTM, but can be generalized straightforwardly to* k*-DTMs.)*

- *They do not go left to* $\triangleright$, *and it is never overwritten: for every* $q \in Q$,

$$\delta(q, \triangleright) = (q', \triangleright, \rightarrow) \text{ or } \delta(q, \triangleright) = (q', \triangleright, \dot{-}).$$

- *Also* $\triangleright$ *is not written otherwise: if* $c \neq \triangleright$, *then* $\delta(q, c)$ *does not output* $\triangleright$.

- *They stop at the accepting and rejecting states: for every* $c \in \Gamma$,

$$\delta(q_Y, c) = (q_Y, c, \dot{-}) \quad \text{and} \quad \delta(q_N, c) = (q_N, c, \dot{-}).$$

Given any reasonable description of δ, *this is easy to check algorithmically.*

Example 4.1. *A deterministic finite automaton* $A = (Q, \Sigma, \delta, q_0, F)$ *is (almost) a partial case of a 1-DTM. Namely, if* $\delta(q, a)$ *is determined, the Turing machine does the same*

$$\delta'(q, a) = (\delta(q, a), a, \rightarrow),$$

that is, it reads the same symbol and writes it back to the tape, and it changes its state similarly to A. *Also*

$$\delta'(q_S, \triangleright) = (q_0, \triangleright, \rightarrow)$$

and for every $f \in F$, *the Turing machine should check by itself whether this is the end of the input:*

$$\delta'(f, \_) = (q_Y, \_, \dot{-}).$$

For non-accepting states $p \notin F$ *the end of the input leads to the reject:*

$$\delta'(p, \_) = (q_N, \_, \dot{-}).$$

The resulting Turing machine works with the same input alphabet Σ, *its tape alphabet* $\Gamma = \Sigma \cup \{\triangleright, \_\}$, *its set of states* $Q' = Q \cup \{q_S, q_Y, q_N\}$, *and its program is* δ'. *It accepts the same language as* A; *moreover, if* A *does not accept a word, our Turing machine rejects it explicitly (comes to* q_N*).*

Example 4.2. *Slides for Lecture 6 contain animated examples of programs for two DTMs: a* 1*-DTM for the majority function and a* 2*-DTM for the naïve pattern matching algorithm from Lecture 1 (but now the pattern is given together with the source).*

Configurations. A configuration describes a "snapshot" of a DTM at a specific point of time (step). It contains the state, the head(s) positions, and the contents of all the tapes (except for the blank space at the end of the tape(s)). For a DTM, its configuration fully determines its next step.

Definition 4.2. *A configuration of a 1-tape DTM is an ordered triple (q, m, p), where*

$q \in Q$ *is the current state,*

$m \in \Gamma^*$ *is the current content of the tape (till the last non-empty symbol),*

$p \in \mathbb{N} \cup \{0\}$ *is the current position of the head.*

For k-tape machines, m and p are vectors.

A run of a DTM. Similarly to finite automata, we define a relation $\vdash$ that tells us what the next configuration will be, i.e., it defines a single step of the machine.

The run of a DTM can be described as a sequence of configurations $C_0 \vdash C_1 \vdash \ldots \vdash C_T$ starting in the "starting configuration" (the state is q_S, the input is on the input tape, all the other tapes are empty, the heads are in the leftmost position observing the symbol $\triangleright$) and finishing in a configuration that contains the state q_Y or the state q_N. It may happen that a DTM does not stop at all, then this sequence is infinite.

Let us give formal definitions for a single-tape machine (they generalize straightforwardly to the case of k-DTM).

A deterministic Turing machine M on input $x \in \Sigma^*$

- Starts in the initial configuration $(q_S, \triangleright x, 0)$:
  - the state q_S, the heads are on the left,
  - the input x is on the first tape (input tape): $\triangleright x_1 x_2 \ldots x_n \_ \_ \_ \ldots$,
  - the other ("auxiliary") tapes are empty ($\triangleright \_ \_ \_ \ldots$),

  and then makes the first step (let us count the steps!).
- At each step, it applies δ.
  Let us define the transition relation $\vdash$:

  Definition 4.3. $(q, m, p) \vdash (q', m', p')$ *is true if and only if*
  - $q' = \boldsymbol{\delta_1}(q, m[p])$ *($m[p]$ is the cell being read/written by the head),*
  - $m'[p] = \boldsymbol{\delta_2}(q, m[p])$ *and* $m'[r] = m[r]$ *for every* $r \neq p$ *(only $m[p]$ is changed),*
  - $p' = p + \boldsymbol{\delta_3}(q, m[p])$ *(where $\rightarrow$ is $+1$ and $\leftarrow$ is -1, and $\div$ is 0).*

  We have just defined $(q, m, p) \vdash (q', m', p')$.
  Let us define $(q, m, p) \vdash^{\boldsymbol{T}} (q'', m'', p'')$ if we can do it in $\boldsymbol{T}$ steps:

$$(q, m, p) \vdash (q^{(1)}, m^{(1)}, p^{(1)}) \vdash (q^{(2)}, m^{(2)}, p^{(2)}) \vdash \ldots \vdash (q^{(i)}, m^{(T)}, p^{(T)}) = (q'', m'', p'')$$

Finally, $(q, m, p) \vdash^* (q'', m'', p'')$ if we can do it in any number $T \geqslant 0$ of steps. That is, the symbol $\vdash^*$ denotes the reflexive and transitive closure of $\vdash$.

A formal definition of the result of computation of a DTM — and its running time. M stops when it reaches q_Y or q_N, so what is its answer? DTMs can solve decision problems and can also solve search problems (essentially, in this case it will compute a function, because its behaviour is deterministic).

If we are talking about <u>decision</u> problems,
For a machine M and an input x, we write:

1. $M(x) = 1$ or $M(x) =$ "yes" iff $(q_S, \triangleright x, 0) \vdash^* (q_Y, \ldots, \ldots)$ (M accepts x),
2. $M(x) = 0$ or $M(x) =$ "no" iff $(q_S, \triangleright x, 0) \vdash^* (q_N, \ldots, \ldots)$ (M rejects x),
3. $M(x) = \nearrow$ iff M does not stop (none of 1, 2 are true).

If we are talking about <u>search</u> problems:
For a machine M and an input x, we write:

- $M(x) = z$ (if $(q_S, \triangleright x, 0) \vdash^* (q_Y, \triangleright z_1 z_2 \ldots z_m, \ldots)$)
  (for a k-tape machine we can designate a specific "output tape"),
- $M(x) =$ "no" (if $(q_S, \triangleright x, 0) \vdash^* (q_N, \ldots, \ldots)$
  (there are no solutions),
- $M(x) = \nearrow$ (if M does not stop).

If M stops on x in T steps, i.e. $(q_S, \triangleright x, 0) \vdash^T (q_Y, \ldots, \ldots)$ or $(q_S, \triangleright x, 0) \vdash^T (q_N, \ldots, \ldots)$, then we say that the running time of M on x is T,

$$\text{time}_M(x) = T.$$

Sometimes we also use the notation $M(x)$ as an abbreviation for "M on input x" (as in: "$M(x)$ stops in at most $|x|$ steps").

Remark 4.1. *In most cases we will be interested in upper bounds on the running time ($\text{time}_M(x) \leqslant T$), because proving lower bounds is more natural for unknown machines and is usually very hard.*

A note on a different notation for configurations — using strings. For a single-tape machine, it may be convenient to write

$$(q, m[0]m[1]\ldots m[s], p)$$

as a string

$$m[0]m[1]\ldots m[p-1]\,\boldsymbol{q}\,m[p]\ldots m[s]$$

(the state q symbolizes the head position; of course, we assume that the tape alphabet Γ and the set of states Q do not intersect).

That is, we start in the configuration

$$q_S \triangleright x_1 x_2 \ldots x_n$$

It may be convenient to add space at the moment we need it

$$\triangleright\, m[0]m[1]\ldots m[s]\,\boldsymbol{q}\,\_$$

so that there is always at least one more symbol after the "head".

It is even more convenient to add a new symbol \$ to denote the (temporary) end of the tape:

$$m[0]m[1]\ldots m[p-1]\,\boldsymbol{q}\,m[p]\ldots m[s]\$$$

One can define $\vdash$ by substitutions in the configuration string using the following "rules" derived from δ:

$$\begin{array}{rcll} qc & \to & q'c' & \text{if } \delta(q,c) = (q', c', \doteq), \\ bqc & \to & q'bc' & \text{if } \delta(q,c) = (q', c', \leftarrow), \\ qcd & \to & c'q'd & \text{if } \delta(q,c) = (q', c', \rightarrow) \text{ and } d \neq \$, \\ qc\$ & \to & c'q'\_\,\$ & \text{if } \delta(q,c) = (q', c', \rightarrow). \end{array}$$

(Such a rule is applied to a substring of the configuration.)

Some of the configurations written this way will be "equivalent": those that differ exclusively by the amount of blank space at the end. One can use certain tricks to avoid this (you can think about some of them as an exercise).

4.2 The binary alphabet

As we know, every text can be encoded in the binary alphabet — otherwise how would it be stored in the memory of our computers. No surprise that one can encode every problem in this alphabet and transform Turing machines working in a different alphabet into Turing machines working over the input alphabet $\{0, 1\}$. Moreover, one can get rid of any symbols in the tape alphabet as well (except for $\{0, 1, \triangleright, \_\}$). The new machine M_2 will achieve the same result on the encoded input $f(x)$ as the old machine M did on the input x. That is, M_2 accepts iff M accepts, M_2 rejects iff M rejects, and M_2 outputs $f(w)$ iff M outputs w.

It is intuitively clear how to do it: one needs to replace every symbol of a different alphabet by a bit string of an appropriate length d and replace the instructions of the original machine so that the new machine will read and write bit strings instead of single symbols. More precisely, one should select the binary logarithm of the original alphabet size as d, and it may be convenient also to encode the blank space symbol _ by d blank symbols so that every "old" symbol of the original machine is replaced by a string of the same size on the tape of the new machine. (The symbol $\triangleright$ is an exception: in never leaves the starting position and we use it for this position only.)

In what follows, we give an overly detailed proof of this statement so that one can use it also as an example of constructing a deterministic Turing machine.

Theorem 4.1. *Consider a deterministic Turing machine M over an input alphabet Σ using a tape alphabet Γ. Let $\Sigma' := \Gamma \setminus \{\triangleright, \_\}$. Let $d = \lceil \log_2 |\Sigma'| \rceil$.*

Consider any encoding f of Σ' by strings of d bits, that is, an injective function $f : \Sigma' \to \{0,1\}^d$ that generalizes to Σ'^* straightforwardly: for $x_1, \dots, x_n \in \Sigma'$, define $f(x_1 x_2 \dots x_n) = f(x_1) f(x_2) \dots f(x_n)$.

Then there is another deterministic Turing machine M_2 working over the input alphabet $\Sigma_2 = \{0,1\}$ using the tape alphabet $\Gamma_2 = \{\triangleright, \_, 0, 1\}$ such that for every $x \in \Sigma^*$, we have $M_2(f(x)) = M(x)$ for yes/no/$\nearrow$ results and $M_2(f(x)) = f(M(x))$ if the answer is a string. Moreover, $time_{M_2}(x) = O(time_M(x))$, where the constant in O depends on d only.

Proof. For the ease of presentation we present the construction for a 1-DTM. It can be generalized to k-DTMs in an obvious way.

We construct δ_2 for M_2 based on δ for M. In addition to the states of the old machine, the new machine will have many more states: we "clone" every state $q \in Q$ into several states keeping also the original state as well.

Consider two states $q, p \in Q$ of the machine M and two symbols $a, c \in \Sigma'$. Let us think about an instruction $(q, a) \mapsto (p, c, i)$ (that is, $\delta(q, a) = (p, c, i)$) of M. To simulate it, our machine will start in q, perform a certain amount of work (in particular, it will recognize a from $f(a)$ and write $f(c)$), and then come to p.

1. We will "recognize" the encoding of $a \in \Sigma$ from its bit representation by introducing new states q^r_s (where $s \in \{0,1\}^*$, $|s| \leqslant d$, and $q^r_\varepsilon = q$) and instructions

$$(q^r_s, b) \mapsto (q^r_{sb}, b, +1) \tag{1}$$

   for every bit $b \in \{0,1\}$ and every string s with $|s| < d$. That is, we come to the state q^r_s after we read s from the tape (the letter r in q^r is for "***r***ecognizing", not for a variable).

   Thus while M executes $\delta(q, a)$ for the symbol a that it reads from the tape, the machine M_2 will come from the state q to the state q^r_α such that $\alpha = f(a)$ is the string of d bits written to the right of the head instead of a. No tape data is changed so far by any of the machines.

   These states and instructions will be in use for every instruction $(q, a') \mapsto \dots$ of M, where $a' \in \Sigma'$: they do not depend on a specific value of a at all (as well as on p, c, i).

2. To simulate the main part of the instruction $(q, a) \mapsto (p, c, i)$ of M (the transition to the new state) we add to M_2 the instruction

$$(q^r_\alpha, z) \mapsto (p^{w,i}_\sigma, z, -1) \tag{2}$$

   if $\delta(q, a) = (p, c, i)$, where $\alpha = f(a)$ and $\sigma = f(c)$. We repeat for every $z \in \Sigma' \cup \{\_\}$. In particular, we add the state $p^{w,i}_\sigma$ to M_2 (more on such states below).

   This instruction is specific for $(q, a) \mapsto (p, c, i)$: it uses all of q, p, c, a, i.

3. We add the new states $p^{m,i}$ and $p^{w,i}_s$ (where $s \in \{0,1\}^*$, $|s| \leqslant d$, and $p^{w,i}_\varepsilon = p^{m,i}$) (the letter w in $p^{w,i}_s$ is for "***w***riting", s is the remaining bit string to be written, and $i \in \{-1,+1,0\}$ is the direction that we remember from the instruction).

   We proceed to writing the encoding of the new symbol c replacing a: we add the instruction

$$(p^{w,i}_{sb}\,, z) \mapsto (p^{w,i}_s\,, b, -1) \tag{3}$$

   to M_2 for every $b \in \{0,1\}$, every $z \in \Sigma' \cup \{\_\}$, and every $s \in \{0,1\}^*$, where $|s| < d$.

   Eventually, we finish writing and come to the state $p^{w,i}_\varepsilon$.

   These states and instructions will be in use for every instruction $\ldots \mapsto (p, c')$ of M with $c' \in \Sigma'$, they do not depend on a specific value of c at all (as well as on q, a). We repeat them for every $i \in \{-1, 0, 1\}$.

4. From $p^{w,i}_\varepsilon$, we proceed to moving the head to where it should move. First of all, we change the state

$$(p^{w,i}_\varepsilon\,, z) \mapsto (p^{m,\,d\,\cdot\,i}\,, z, 0) \tag{4}$$

   (for every $z \in \Sigma' \cup \{\_\}$). The head now points to the first bit of the string we have just written. We introduce new states $p^{m,j}$ for every j, where $-d \leqslant j \leqslant d$, where the letter m means "***m***oving" and j is the remaining number of moves with the corresponding sign i (initially, $j = d \cdot i$).

   To move the head, we add the instruction

$$(p^{m,\,j}\,, z) \mapsto (p^{m,\,j'}\,, z, i') \tag{5}$$

   for every $j \neq 0$, where $-d \leqslant j \leqslant d$, $j \neq 0$, $i' = \operatorname{sign}(j)$, $j' = j - i'$. We do it for every $z \in \Sigma' \cup \{\_\}$ (the case of _ will be used later).

   These states and instructions do not depend on q, a, and c, they are introduced for every state p. The instruction (5) also does not depend on i. There is no harm in repeating the instruction (4) for all the three values of i.

5. Eventually, we add the instruction

$$(p^{m,\,0}\,, z) \mapsto (p\,, z, 0). \tag{6}$$

   for every $z \in \Sigma' \cup \{\_\}$ to get to p after all this work is done.

   We add this instruction for every state p.

6. In total, what happens when M_2 executes all these instructions:

   - it reads the bit representation of a,
   - it writes the bit representation of b,
   - it moves the head to the beginning of the bit representation of the symbol that M would see at the next step.

At the moment this is not absolutely accurate, because we did not say what to do with the special symbols $\triangleright$ and _ from the tape alphabet. We treat this case below.

7. For the symbol _, for the ease of presentation we replace it by d symbols _ so that it would take the same amount of space as any symbol in Σ'. In order to be able to recognize and to write the description of _, we add new states q_s^r and $p_s^{w,i}$ for $s \in \{\_, \ldots, \_^d\}$ and add the instructions

$$\begin{aligned} (q_s^r\,, \_) &\mapsto (q_{s\_}^r\,, \_, +1). \\ (p_{s\_}^{w,i}\,, z) &\mapsto (p_s^{w,i}\,, b, -1) \end{aligned}$$

for every $s \in \{\varepsilon, \_, \ldots, \_^{d-1}\}$, $p, q \in Q$, and $i \in \{-1, 0, +1\}$.

We also add a version of the instruction (2) with α replaced by $\_^d$ and σ replaced by $\_^d$ if M contained the respective instructions $(q, \_) \mapsto \ldots$ and $\ldots \mapsto (p, \_, i)$.

8. Eventually, for the symbol $\triangleright$, we leave it as is. The only two cases where M worked with it were:

   (a) An instruction of the type $(q, \triangleright) \mapsto (p, \triangleright, +1)$. We include it into the program of M_2 without any change.

   (b) If M had to move its head left, and there was $\triangleright$ there. Note that (5) would not work here, so we add the following version of it:

$$(p^{m,\,j}\,, \triangleright) \mapsto (p^{m,\,0}\,, \triangleright, 0) \tag{7}$$

   for every $p \in Q$ and every j, where $-d + 1 \leqslant j \leqslant -1$.

□

Remark 4.2. *If one would like to change the input alphabet only without touching the tape alphabet, it can also be done with $d = \lceil \log_2 |\Sigma| \rceil$. For example, one can encode the other symbols of the tape alphabet by appending $d - 1$ blank spaces _ to them like we did with the symbol _ above.*

We will keep it in mind in the future that we can always think about the binary alphabet and can always reencode everything into it. Thus when it is more convenient for us to write programs for Turing machines using a different alphabet, we will do that and still think that we could do it with a binary alphabet if necessary.

4.3 The number of tapes

It turns out that

- the number of tapes does not affect the principal power of DTMs: machines with 1 tape and with $k = O(1)$ tapes solve the same problems,

- even the running time does not change too much (namely, it changes quadratically),
- this quadratic time difference is essential, and we will demonstrate it later using the language of palindromes: it can be solved in $O(n)$ steps on a 2-DTM, but it requires $\Omega(n^2)$ steps on a 1-DTM,
- for machines with $k \geqslant 2$ tapes it changes at most by a logarithmic factor (we will not prove that — it is a bit more complicated), so only the case of single-tape machines is a little bit different.

Theorem 4.2. *For every k-DTM K there exists a 1-DTM M that produces the same answer, that is, for every input x, $M(x) = K(x)$. Moreover, $time_M(x) = O((time_K(x) + |x|)^2)$. The constant in $O(\dots)$ depends on K but does not depend on x.*

Remark 4.3. *To be absolutely formal (or even formalistic), one can consider the properties of M as three statements:*

1. *If we consider decision problems, M accepts x iff K accepts x, M rejects x iff K rejects x.*
2. *If we consider search problems, M returns the same solution for x as K, and if K says that there is no solution, so does M.*
3. *In any case, $time_M(x) = O((time_K(x) + |x|)^2)$. In particular, M stops on x iff K stops on x.*

Proof. Let Γ_K be the tape alphabet of K and δ_K be its transition function.

The program of M will be a modified program of K. We will keep the content of all tapes of K on the (single) tape of M:
cells $0, 1, 2, \dots, i, \dots$ of tape 1 of K are mapped to the cells $1 + 0 \cdot k,\ 1 + 1 \cdot k,\ 1 + 2 \cdot k, \dots, 1 + i \cdot k, \dots$,
$\dots$
cells $0, 1, 2, \dots, i, \dots$ of tape j of K are mapped to the cells $j + 0 \cdot k,\ j + 1 \cdot k,\ j + 2 \cdot k, \dots, j + i \cdot k, \dots$.

We will expand the alphabet. Namely, for every old symbol $a \in \Gamma_K$ we will add its "marked" copy $\bar{a}$ that will mean "a is here and the head of this tape is here". We will also add a new symbol $\triangleright'$ and its marked copy $\bar{\triangleright'}$ in order to keep the whole content of each tape of K including the starting cell (we cannot use the original $\triangleright$ more than once by our definition). In total,

$$\Gamma = \Gamma_K \cup \{\bar{a} : a \in \Gamma_K \setminus \{\triangleright\}\} \cup \{\triangleright', \bar{\triangleright'}\}.$$

Actually, to facilitate some of the constructions below we will introduce more symbols on the fly, but their number will not depend on x. The same applies to additional symbols that may be needed to prepare the tape initially and for the final cleanup.

Somewhat similarly to the proof of Theorem 4.1, for each state $q \in Q_K$ of the original machine, we will introduce several additional states and add them to the set of states Q of the new machine (we will keep the old states as well, except that we will refer to the old initial, accepting and rejecting states as q_{SK}, q_{YK}, q_{NK} — they will not be initial, accepting, or rejecting anymore).

Before simulating the computation of K, our machine M prepares the description of its initial configuration. That is, It spreads the input to the cells $1 + 1 \cdot k, 1 + 2 \cdot k, 1 + 3 \cdot k, \dots$ (it is an

easy exercise how to do it in quadratic time without corrupting the data) and writes $\bar{\triangleright}'$ to the cells $1 + 0 \cdot k$, $2 + 0 \cdot k$, $\ldots$, $k + 0 \cdot k$ to "put the heads" of K to the initial positions on all the tapes of K.

After these preparations, eventually we transit to the state q_{SK}.

Let us describe the simulation of one step of K. Denote the current state by q. W.l.o.g. our (M's) head is at the starting position $\triangleright$. There are k marked symbols on the tape (those that would be observed by K's heads).

- To simulate the function δ_K of K, we need to find out its second argument: the vector of symbols it reads from its k tapes. Let us find these symbols. It is not difficult: they are marked (as $\bar{a}$ instead of a)!

  We scan the tape left-to-right and collect the k symbols being read. We keep them temporary in our finite memory, that is, in the state. To accomplish this, we introduce many new states $q^r_{a_1,a_2,\ldots,a_k}$ for $a_i \in \Gamma_K \cup \{\varepsilon\}$. Informally, the non-empty symbols in the superscript mean that we have already found and read them; ε stays for the tapes whose symbol is not yet found. So we perform the transition $(q, \triangleright) \vdash (q_{\varepsilon,\ldots,\varepsilon}, \triangleright, \rightarrow)$ and continue going right until we find one of the symbols a, then we transit to the corresponding state $q^r_{\varepsilon,\ldots,a,\ldots,\varepsilon}$ and continue until we find all the symbols.

  Caveat: Note that we find these symbols in an unknown order, maybe a_3 first, then a_1, etc. To know the index (subscript) of $q^r_{\ldots}$ that we should put the symbol two, we can use, for example, additional states $q^{r,i}_{\ldots}$ for $1 \leqslant i \leqslant k$ and update the upper index cyclically as we go right.

- Once we have found all the symbols, we know what δ_K of K would produce, assume that

$$\delta_K(q, (a_1, \ldots, a_k)) = (p, (b_1, \ldots, b_k), (d_1, \ldots, d_k).$$

  Then perform the transition

$$(q^r_{a_1,\ldots,a_k}, c) = (p^{wm}_{b_1,b_2,\ldots,b_k;d_1,\ldots,d_k}, c),$$

  where $p^{wm}_{\ldots}$ are new states for writing b_i's and moving the heads according to d_i's; all a_i are here in Γ_K, there are no ε's.

  To be more precise, we better make additional copies $p^{wm,i}_{\ldots}$ like we did with q. Therefore, the actual transition would be from $q^{r,i}_{a_1,\ldots,a_k}$ to $p^{wm,i}_{b_1,b_2,\ldots,b_k;d_1,\ldots,d_k}$. We introduce the same number of new states $p^{wm,i}$ for $b_1, b_2, \ldots, b_k \in \Gamma_K \cup \{\varepsilon\}$ like we did with q^r, also $d_1, \ldots, d_k \in \{-1, 0, +1\}$.

- Now we scan the tape right-to-left (it suffices to do it starting at the position where we have found the last symbol a_i) and write the k symbols back to the tape in the same way as we read them. Again, to find out positions where a symbol is to be replaced, we look at marked symbols $\bar{c}$; we also use the superscript i and the subscript b_i in order to determine what to write. So far we write $\bar{b_i}$, not b_i itself.

- After we finish this stage in the state $p^{wm,i}_{\varepsilon,\dots,\varepsilon;d_1,\dots,d_k}$ for some i, it is time to move the heads. We transit to the state $p^{m,i}_{d_1,\dots,d_k}$ — we introduce these states similarly to the previous ones, here $d_1,\dots,d_k \in \{-1,0,1,\varepsilon\}$. We scan the tape left-to-right, we find again marked symbols, and we move the respective head in the respective position. To do that, we replace some c with $\bar{c}$ and vice versa (note that we do it with a gap of k positions, because the head is moved on one of the old tapes, so perhaps we will need to introduce more auxiliary states to be able to count to k).

- When we finish with all this, we get to the state $p^{m,i}_{\varepsilon,\dots,\varepsilon}$, transit to the state p and go left until we find $\triangleright$ — see item 4.3.

We need to write some instructions also for q_{YK} and q_{NK}. If our machine decides a decision problem, we could simply assume $q_Y = q_{YK}$ and $q_N = q_{NK}$, and finish the computation.

However, if our machine decides a search problem (computes a function), we need to write down the output. It is already there, but it is written with gaps of k symbols and with some garbage between the symbols (and also with $\bar{c}$ instead of c at the head position). It is easy to perform this conversion (in particular, remove the garbage between the cells) in quadratic time. The only caveat is that we do not know where the end of tape is (so far we did not need to know it). To cope with this, once can introduce a new symbol \$ marking the end of the representation of all the tapes and move it when one of the heads tries to surpass it.

Running time. We simulated each step of K using one left-to-right and one right-to-left scan. The total number of cells in one scan is at most $k \cdot (\mathrm{time}_K(x) + |x|) + 1$, because K had no time to write more symbols thatn $\mathrm{time}_K(x)$. We are never staying at any position longer than for a small constant number of steps. Only when we "move the heads" we can move k symbols back and force, it would add $k^2 \cdot const$ steps to each scan.

Conversions of the input and of the output take quadratic time in the length of the input $|x|$ and the length of M's tape, respectively.

Therefore, the total time of M is $\mathrm{time}_M(x) = O((\mathrm{time}_K(x) + |x|)^2)$, where the constant in O may depend on K (in particular, on the number k of its tapes). □

Remark 4.4. *It is easy to see that this proof is constructive, i.e., one can write an algorithm that transforms a description of K into a description of M. The same applies to the alphabet change in Theorem 4.1. We will use this fact later after we define a specific format (encoding) for describing Turing machines.*

4.4 A time lower bound for 1-tape DTMs

The quadratic time difference between 1 and 2 tapes is essential, which can be demostrated using the language of palindromes: it can be solved in $O(n)$ steps on a 2-DTM, but it cannot be solved much faster than in n^2 steps on a 1-DTM.

Example 4.3 (Palindromes)**.** *Consider $L = \{x \in \Sigma^* : x = x^R\}$ over a finite alphabet Σ. Let us build a 2-DTM for it, $Q = \{q_S, q_Y, q_N, q_1, q_2\}$. The idea is as follows (all the instructions should be repeated for each $z \in \Sigma$):*

1. *Copy (symbol by symbol) the input to the second tape.*

$$\begin{aligned}\delta(q_S,(a,z)) &= (q_S,(a,a),(\rightarrow,\rightarrow)) \text{ for every } a\in\Sigma\\ \delta(q_S,(\_,z)) &= (q_1,(\_,z),(\leftarrow,\leftarrow))\end{aligned}$$

2. *Now the head on the second tape is looking at the last symbol.*
   Return the head of the first tape to the start and then to the first symbol of the input.

$$\begin{aligned}\delta(q_1,(b,z)) &= (q_1,(b,z),(\leftarrow,\dot{-})) \text{ for every } b\in\Sigma\\ \delta(q_1,(\triangleright,z)) &= (q_2,(\triangleright,z),(\rightarrow,\dot{-}))\end{aligned}$$

3. *Compare (symbol by symbol) the contents of the tapes, while the head on the first tape is going left-to-right and the head on the second tape is going right-to-left. We must finish on the start cell of tape 2 and a blank symbol on tape 1 simultaneously.*

$$\begin{aligned}\delta(q_2,(c,c)) &= (q_2,(c,c),(\rightarrow,\leftarrow)) \text{ for every } c\in\Sigma\\ \delta(q_2,(\_,\triangleright)) &= (q_Y,(\_,\triangleright),(\dot{-},\dot{-}))\\ \delta(q_2,(d,e)) &= (q_N,(d,e),(\dot{-},\dot{-})) \text{ for every } d\neq e \text{ except for the pair } (d,e)=(\_,\triangleright)\end{aligned}$$

It is obvious that the number of steps is $O(n)$.

We prove that every 1-DTM must take a quadratic number of steps in order to distinguish palindromes from non-palindromes.

Reminder: $f(n)=\boldsymbol{\Omega}(g(n))$ means that $\exists\varepsilon>0\ \exists N\ \forall n>N \quad f(n)\geqslant\varepsilon\cdot g(n)$

Theorem 4.3. *Let $\Sigma=\{0,1\}$. For every 1-DTM M giving a correct yes/no answer about the language in Example 4.3, there is an infinite sequence of words $w_n\in\{0,1\}^{3n}$ ($n\in\mathbb{N}$) such that M must perform $\Omega(n^2)$ steps on w_n.*

Proof. Assume that a 1-tape DTM M does the job. W.l.o.g. assume that its head always moves $\rightarrow$ or $\leftarrow$ and never stays in the same position (this assumption could increase the time at most twice).

Let us consider the behaviour of M on inputs w_n of the form $x0^nx^R$, where $x\in\{0,1\}^n$. Let T_n be the number of steps it takes on w_n.

We will look at a specific position i on the tape and consider moments when the head crosses the boundary between the cells i and $i+1$: after the head moves to the right from the cell i *and* after it comes back to i back from the right side. Let us define what is a "crossing sequence" at the position i: this is the sequence $(q_1,\ q_2,\ \ldots)$ of the current states at these moments (odd-numbered states correspond to moves to the right).

Since at each step M crosses some boundary and there are n indices in the middle substring 0^n of w_n, one of the crossing sequences on 0^n is shorter than $\frac{T_n}{n}$. Let us fix this specific position i for the rest of the proof.

A crucial observation is that similar crossing sequences lead to similar behaviour, as formulated in the following lemma.

Lemma 4.1. *Let $n+1 \leqslant i \leqslant 2n$. Let $x, x' \in \{0,1\}^n$. If both $x0^n x^R$ and $x'0^n x'^R$ lead to the same crossing sequence at i, then M accepts $x'0^n x^R$.*

Proof. Consider three runs: on inputs $x0^n x^R$ (which we will call "the green input") and $x'0^n x'^R$ ("the red input"), and the "mixed" input $x'0^n x^R$.
Note that all the changes in the cells right to i are made between two consecutive crossings (to the right and to the left) are fully determined by the content of these cells and the state when the head enters this space. (The same holds, symmetrically, for the cells $\leqslant i$ between the left and the right crossings.)
Let us prove the following statement (and thus the lemma) by induction on the number t of crossing (we assume $t = 0$ at the start of the computation):

1. Till crossing t cells $\leqslant i$ of the mixed run stay as in the red run, and cells $\geqslant i+1$ stay as in the green run.

2. If t is even, between q_t and q_{t+1} the state and the head position of the mixed run are as in the red run;
   If t is odd, they are as in the green run.

Base: From the beginning till the first crossing $t = 1$ the statement is true.

Step: We consider the behaviour of M from one odd crossing, number t, to the next odd crossing, number $t+2$.
By the induction hypothesis red, mixed and green run came to the odd crossing t in the same state q_t as $(q_t, \triangleright u'v', i+1)$ and $(q_t, \triangleright u'v, i+1)$ and $(q_t, \triangleright uv, i+1)$, respectively; or, in another notation, as $\triangleright\, \boldsymbol{u'\, q_t\, v'}$ and $\triangleright\, \boldsymbol{u'\, q_t\, v}$ and $\triangleright\, \boldsymbol{u\, q_t\, v}$, where $|u| = |u'| = i$ and $|v| = |v'| = 3n - i$. Note that there are no zeroes (0^n) between u and v — the content of the tape has already changed, and the point is that the left part (u') stays in sync for the red and the "mixed" runs, and the right part (v) stays in sync for the green and the "mixed runs.

What happens till the next odd crossing $t+2$?

- First, **the mixed and the green runs work in sync on the same string v until we come back to i** from the right in the left direction.

- Thus the mixed run comes back to i for the even crossing $t+1$ in the same state q_{t+1} as the green run. Since the green and the red run share **the same crossing sequence**, this is the same state for the red run as well.

- Meanwhile u (or u') did not change in any run as the head was not there, thus, the configurations of the three runs become
  $\triangleright\, \boldsymbol{u'_{1..i-1}\, q_{t+1}\, u'_i z'}$ and $\triangleright\, \boldsymbol{u'_{1..i-1}\, q_{t+1}\, u'_i z}$ and $\triangleright\, \boldsymbol{u_{1..i-1}\, q_{t+1}\, u_i z}$.

- After that, all the runs work on the cells at positions $\leqslant i$. In this period the mixed and the red runs are in sync as they work on the same content u' until the next odd (right) crossing $t+2$. □

Now we are back to the proof of the theorem. For any two different n-bit strings $x \neq x'$, a correct M should not accept $x'0^n x^R$. Therefore, x can be described by i and the corresponding crossing sequence: it is impossible that $x \neq x'$ share the same description. Such a description

can be encoded using $O(\log_2 n + \frac{T_n}{n})$ bits (the logarithm is for describing i). However, there are 2^n strings x and only at most $2^{(\log_2 n + \frac{T_n}{n}) \cdot const}$ such descriptions; necessarily, the number of descriptions must be at least 2^n, that is,

$$\begin{aligned} (\log_2 n + \tfrac{T_n}{n}) \cdot const &\geqslant n \\ \tfrac{T_n}{n} &\geqslant \tfrac{n}{const} - \log_2 n \\ T_n &\geqslant \tfrac{n^2}{const} - n \log_2 n \quad = \Omega(n^2) \end{aligned}$$

□

4.5 A note of data representation (encoding)

Our plan is to build a Universal Turing machine U that would take on input the description of another Turing machine M and the input x of M and simulate the behaviour of M on x.

In order to talk about it, we need to choose the representation of it: how do we write a Turing machine as a string in a finite alphabet. So a convenient time came to finally talk about data representation, called also encoding, concerning not just Turing machines.

In what follows, we consider reasonable encodings for the inputs (as well as the outputs, if any). What does it mean?

For example, if a decision problem talks about graphs, a graph may be given on input as a binary word in a way that allows to check quickly (= in polynomial time) whether it contains a specific edge. On the other hand, the encoding should not be excessive (= containing exponentially many bits where a polynomial number of bits would suffice) or "fantastic" (containing information that may be difficult to compute from the mathematical formulation, for example, the answer to the problem).

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 usually we will omit them.

4.6 Encodings of Turing machines. Turing numbers

Let us choose a specific reasonable encoding of Turing machines. For clarity, let us start with the alphabet $\{\#, 1\}$. The encoding of a DTM will include

- the number of tapes, in unary: 1^k, then #,
- the number of states, in unary: $1^{|Q|}$, then #,
  states will be encoded also in unary
  (we assume that q_S, q_Y, q_N are encoded by 1, 11, 111, respectively),
- the size of Γ, in unary: $1^{|\Gamma|}$, then ###
  symbols will be encoded also in unary
  (we assume that $\triangleright$ is encoded by 1 and _ by 11),

- instructions, one by one, as
  $\langle q\rangle\#\#\langle a\rangle\#\#\langle p\rangle\#\#\langle b\rangle\#\#\langle d\rangle\#\#\#\#$
  if $\delta(q,a) = (p,b,d)$ (where $\langle d\rangle \in \{1, 11, 111\}$),
- if there are k tapes, then we write $\langle a_1\rangle\#\ldots\#\langle a_k\rangle$ instead of a (respectively, b and d),
- add any number of additional #'s at the end.

We can replace # by 0 and get a natural number!

Thus one can attach a Turing number to every DTM (actually, every machine gets infinitely many Turing numbers because of the #'s at the end; it may be useful in what follows).

Some of the natural numbers do not follow this syntax; we map syntactically wrong numbers to the trivially rejecting machine $(q_S, \triangleright) \to (q_N, \ldots)$, thus every number correspond to some machine.

In what follows, when we say "its Turing number", *sometimes* we mean the smallest one without further notice (usually, it does not matter).

4.7 A Universal Turing machine

It turns out that all DTMs can be simulated on a single specific machine (called universal Turing machine, UTM); the input of this machine is a description of a simulated machine and its input. It is not a surprise: this is what interpreters, compilers, and operating systems do. One can certainly implement a C compiler in the C language... After all, this is what the Church–Turing thesis suggests; we will, however, build a UTM ourselves and estimate its running time (or, put another way, overhead over the running time of the simulated machine).

After proving that, when we design DTMs, we can do it using a convenient (finite, i.e., constant) number of tapes and alphabet; moreover, we can let DTMs run other DTMs by the Turing number. When we have to analyze (unknown) DTMs, we can assume that they have just the tapes needed for the formulation and that they work over $\Sigma = \{0, 1\}$. We will construct a universal DTM with more than one tape; by Theorems 4.2 and 4.1 we can convert it further to a 1-tape machine with a very limited tape alphabet. In fact, in the rest of this course the only important thing is that this simulation take polynomial time; that is, the time spent by the UTM on a machine M and an input x is bounded by $p(t + n)$, where p is the time spent by M on x, n is the length of x, and p is a polynomial. In fact, the polynomial will be a very small one, and the only its component depending on M is a multiplicative factor. Its degree does not depend on M and, of course, its degree or its coefficients do not depend on x.

Theorem 4.4. *Denote by $\langle M, x\rangle$ a specific easily parseable reasonable binary encoding of the pair consisting of a DTM M (represented by a Turing number $\langle M\rangle$) and an input $x \in \{0, 1\}^*$ (for example, $\langle M, x\rangle = 1^{|M|}0\langle M\rangle x$).*

There exists a 2-DTM U such that
for any (multitape) DTM M for $\{0, 1\}$ and
for any string $x \in \{0, 1\}^*$,

- $U(\langle M, x\rangle) = M(x)$.
- *In particular,* $U(\langle M, x\rangle)$ *stops iff* $M(x)$ *stops.*
- *Moreover,* $time_U(\langle M, x\rangle) = O((time_M(x) + |x|)^2)$*, where the constant in* O *depends on* $|\langle M\rangle|$.

If completing some formal details of the following construction seems too difficult, one can think about additional tape(s): it is not very important for us to have a 2-tape (and not, say, 9-tape) UTM.

Proof. The tape alphabet Γ_U of our machine will contain $\{\triangleright, \_, 0, 1, \#\}$. We will also add auxiliary duplicates of these symbols like $\hat{0}$ when needed — they serve us to leave a mark.

First of all, we can parse $\langle M, x\rangle$ into $\langle M\rangle$ and x in linear time and work for some time on $\langle M\rangle$ only. By Theorem 4.2 (or rather its constructive proof) we can compile k-DTM M into a new 1-DTM M_1 that is only quadratically slower. The running time of this transformation from $\langle M\rangle$ into $\langle M_1\rangle$ does not matter as it depends on $|\langle M\rangle|$ only and not on x — recall that everything depending on M only will contribute to the constant in $O(\ldots)$.

We can compile M_1 into a machine M' with the tape alpabet $\{0, 1, \triangleright, \_\}$ by Theorem 4.1 with a linear slowdown. (Actually, we could live without it, but it simplifies the notation.)

We will keep the memory of M' on the 1st tape. We keep the description of M' and the current state q on the 2nd tape (as $\langle M'\rangle\_\langle q\rangle$).

Initially $\langle q\rangle = 1$, this is the encoding of the initial state of M'.

During the simulation our head on the 1st tape will follow the head of M'.

To simulate a single step of M', do the following:

- Let c be the current symbol on the 1st tape, and q be the current state.
- Scan $\langle M'\rangle$ to match q and c, thus finding an instruction $(q, c) \vdash (q', c', d')$.
- Replace c with c', move the head on the 1st tape in the direction d'.
- Copy the new state q' to where q was.

Before simulating each step, we need to check that q is not q_Y or q_N of M'; in this case we accept or reject immediately.

Observe that we use $O(1)$ steps of U per one step of M'. The pre-processing stage takes linear time (the only thing depending on x amounts to copying it to the proper positions of the 1st tape — everything else depends on M only). □

Remark 4.5. *Our 2-tape UTM performs in particular pattern matching and copying on the 2nd tape containing* $\langle M'\rangle\_\langle q\rangle$*. The size of this tape depends on* $|\langle M\rangle|$ *only and contributes to the constant in* $O(\ldots)$*, therefore the time complexity of these procedures is not important.*

If one does not want to add another working tap to U*, pattern matching can be performed as follows:*

- *Mark the current position in the pattern and in the text by replacing a symbol* c *with a "marked" symbol* $\hat{c}$*.*

- *Use the finite memory (state) to remember what symbol we are matching while moving the head between the pattern and the text (like "q_0 looks for 0" and "q_1 looks for 1").*
- *To make it easier to program, expand the alphabet and use more signs $\hat{c}$, $\bar{c}$, $\dot{c}$, $\ddot{c}$ to mark a place to come back.*

Copying or moving a string to another place can be performed by using the "marks" like $\hat{c}$ instead of c to find again the last written symbol, if unclear.

4.8 RAM: Random Access Memory and Random Access Machine

Turing machines have an obvious (imaginary?) difference from contemporary computers: in order to access data, TMs need to spend time proportional to the relative address. On the contrary, when programming, we think about instant data access: who cares what is the value of the index i when accessing the i-th cell of an array? Even if some cache is kept for accessing extremely recent data, it is limited, so the time needed to access a cell of memory does not seem to depend much on its address.

Random Access Memory is a model of memory access that is close to this (actually, sometimes wrong) intuition. (The word "random" here refers to equal time access and not to probabilistic considerations.) RAM machines (or simply RAM) are a model of computation that uses random access memory instead of tapes. Typically, they resemble an "assembler" (or command) language that is underlying the architecture of contemporary computers, as viewed by a programmer caring for extremely time-critical pieces of software.

The main feature of this model is an "almost instant" read/write access to the data stored in a cell of memory: one uses `mem`$[i]$ to address the data stored in the cell located at the address i, and typically the value of i is also kept in a certain cell (say, `mem`[239]).

4.8.1 Definition: A specific instruction set

The command set may vary. In these notes we use a minimalistic set of commands, namely, the one from Papadimitriou's book [Papadimitriou]. (This subsection about RAM largely follows this book.) The memory model consists of a potentially infinite number of cells `mem`$[i]$, where i is a nonnegative integer number (this model operates on integers). Each cell is capable to store any integer number. For convenience (and resemblance to hardware), the cell `mem`[0] has a special meaning: it is an "accumulator" where most of the operations are done. In addition to the data, this model has a "program counter" κ: the address of the instruction to be executed. A program of RAM is a numbered list of instructions of the following types:

READ reads the data from the d-th argument (part of the input) and writes it to the accumulator;

LOAD copies the data (a constant or the content of some cell) and writes it to the accumulator;

STORE does the opposite thing;

ADD adds the value of the i-th cell to the accumulator;

SUB performs subtraction (similarly);

HALF divides the value of the accumulator by two (for nonnegative numbers, it means one bit shift),

JUMPs are "goto" instructions that influence the order of the execution: they provide the next value c of the program counter κ, and their result is conditioned on the value of the accumulator: JZERO performs a jump if the accumulator is zero; JPOS and JNEG do it if it is strictly positive or negative, respectively; JUMP does it unconditionally.

HALT stops the computation, the output is in the accumulator.

Arguments of these instructions can be immediate (a specific constant c, like "42") direct (a specific address d, i.e., a specific variable, like `mem`[42]), or indirect (a cell addressed by another cell, like `mem`[`mem`[42]]). In the instruction set above i can also stand for d and c, and d can also stand for c, where applicable.

The program starts its execution with $\kappa = 1$ (the first instruction) and empty memory (every cell keeps the value zero) and continues until it encounters the HALT instruction. Such a machine computes a function in a natural way. Depending on the task, one can consider n integer inputs (thus the function will be $\mathbb{Z}^n \to \mathbb{Z}$) or n one-bit inputs (then $\{0,1\}^n \to \mathbb{Z}$).

Example 4.4. *The notation used in the example before is more intuitive than the one used in the slides. The following program computes the majority (the most frequent value) of its input bits. If there is an equal value of zeroes and ones, then it returns zero. The last input register contains* -1 *so that we understand that it is indeed the last one.*

*This program is essentially a "while" cycle: it reads the next input register, adds it (*1 *or* 0*) to the sum, and repeats it until it finds* -1*; then it checks whether the sum is greater than* $n/2$ *(it must count the number of inputs) and halts with the corresponding answer in* `mem`[0].

```
// the counter mem[1] and the sum mem[2] are initialized by 0 by default
// now the while cycle begins
01. READ input number mem[1] // the mem[1]-th input
02. JNEG 07                  // the end of the input?
03. ADD  mem[2]              // mem[0]:=mem[0]+mem[2], the sum is updated
04. LOAD from mem[1]
05. ADD  constant 1          // we increased the counter mem[1]
06. JUMP 01
// the while cycle ends
07. LOAD from mem[1]
08. HALF                     // compute [n/2]
09. SUB  mem[2]              // this is [n/2]-sum
10. JNEG 13                  // if it is negative, we output 1
11. LOAD constant 0
12. HALT
13. LOAD constant 1
14. HALT
```

Example 4.5. *Please refer to lecture slides for a more elaborated example (multiplication of nonnegative integers).*

4.8.2 The running time of a RAM program

It is disputable what is the time taken by a specific run of a RAM program.

1. Certainly one execution of an instruction takes at least one step (unit-cost RAM).

   Pro: This is practical if in our application our integers fit the "word size" of the computer and each instruction indeed costs $O(1)$ time.

   Contra: One can achieve an exponential speedup by using very large integers (in practice it won't happen!).

2. Or should a single step be charged by the number of bits in everything involved in the instructions (values, addresses: i, mem$[i]$, mem[mem$[i]$])? This is called log-cost RAM.

   Pro: This is absolutely fair!

   Contra: It bears an extra theoretical effort if in the reality the integers are relatively "small" and each one fits a single cell (or maybe $O(1)$ cells) of our random-access memory.

A nice feature of the specific command sent above it that it does not matter for it, in particular, because there is no multiplication. The multiplication is easy to implement in this model (an easy exercise — do it yourself or see the slides), but it takes more than just the number of bits in its operands (the "graduate school algorithm" is quadratic-time, one can do better with more complicated algorithms, but still they are not as good as the linear-time algorithm for the addition — of course, no hardware can do better for integers of unbounded size).

Let us prove it in a more formal way.

Lemma 4.2. *Consider a specific RAM program. After t steps of its execution, the number of bits in any memory cell is at most $t + S$, where S is the maximal bit-size of the input and all integers in the program text.*

Proof. Induction on t. At start, the statement is obvious. Consider the situation after $t \geqslant 1$ steps. The only instructions increasing the maximum bit-size of the memory cells are `ADD` and `SUB`. By the induction hypothesis, each of the arguments of the instruction contained at most $t - 1 + S$ bits. These operations can increase the number of bits (of the longest argument) at most by 1. □

Therefore, for our particular set of instructions of RAM there is no much difference between the unit-cost time measure and the log-cost time measure: if a machine makes $t \geqslant S$ steps, then its memory cells never contain more than $2t$ bits, and its log-cost time is at most $2t^2$ — not a big price for switching from a much more realistic to a much easier-to-estimate model! Therefore, in what follows we stick to the unit-cost version: the time spent by our RAM is simply the number of executed commands.

4.8.3 Equivalence to DTMs

Despite of the fact that RAM machines use a more efficient memory model, they are largely equivalent to DTMs.

In order to compare these models, let us assume that they get the input in a similar way: when a DTM receives the string of bits $x_1x_2\dots x_n \in \{0,1\}^n$, a RAM receives the values $x_1,\dots,x_n$ in its first n input registers, and also the value -1 in the next ($(n+1)$-th) register, which marks the end of input.

Let $\mathrm{time}_R(x)$ be the time that R takes on x according to the unit-cost model (that is, the total number of executions of instructions).

Theorem 4.5. *For every RAM program R one can construct a DTM M that computes the same function and takes time $O((time_R(x)+|x|)^3)$ on input x (where R gets $|x|$ bits in $|x|$ input registers and T gets x as an input string of bits).*

Remark 4.6. *The opposite direction is an easy exercise (with some natural representation of bit strings as integers).*

Proof. Let t be the unit-cost time of R on x, and let $T = t + r + n$, where r is the size of the bit representation of R's program (with current definitions it is enough to assume that r is the maximum bit size of an absolute value of any number explicitly mentioned in the arguments of the program).

Our DTM M will keep R's memory on one of its tapes, call it MEM. The tape alphabet will include $:, \#$ and other delimiters if needed. The format for storing the content of a single cell i is i:mem[i]. Such records will be separated by any number of #'s, e.g.,

11 : 10100#####0 : −1#100 : −101001_

which means that mem[3] = 20, mem[0] = −1, and mem[4] = −41.

Given i, in order to find (or store) the value of mem[i], we need to scan the tape once (use any linear-time pattern matching algorithm). Note that by Lemma 4.2 its length is at most $O(T^2)$: at most T cells are ever written, each containing at most T bits.

For convenience, write down R's program on another tape (in principle, we could avoid it and keep all the necessary information in the transition function of the DTM).

We will keep the current instruction number ("program counter", κ) on a yet another tape.

Eventually, the input of RAM (and of DTM) will be kept on tape 1 and will never be overwritten.

To interpret an instruction,

- check its type (thus go to an appropriate state of a DTM),
- retrieve the values of its arguments from MEM to auxiliary tapes by scanning MEM,
- compute the result (note that it takes at most $O(T)$ steps even for ADD and SUB applied to $O(T)$-bit values — an easy exercise),
- write it back to MEM: wipe the old record i:*oldvalue* by writing #'s over it and append the new record i:*newvalue* to the end of tape,

- update the program counter: add +1 if it was not a "jump", otherwise update it according to the type and the argument of the jump instruction.

On `HALT`, copy the value of `mem`[0] to the output tape and stop.

A single execution of R's instruction takes $O(T^2)$ steps of our DTM, thus the total running time is $O(T^3)$. □

This theorem tells us that from now on we can indeed think about our intuitive notion of an algorithm as a DTM (and, of course, vice versa). In particular, we can explain DTM's program in "pseudocode", and when we need to work with algorithms as objects (so-to-say, executable files that one can treat and analyze as binary strings), we can assume that they are DTMs, i.e., programs with very simple syntax and semantics (configurations and the transition relation).

5 Nondeterministic Turing machines

In finite automata we have seen that the deterministic model is equivalent to the nondeterministic model. However, it comes with a price: the number of states may increase exponentially.

For Turing machines we observe a somewhat similar yet differently formulated phenomenon: one can convert a nondeterministic Turing machine into a deterministic one, but the complexity (in this case it is the running time, that is, the number of steps) may increase exponentially.

Nondeterministic Turing machines is not a model of computation corresponding to some physical device. Yet it is an incredibly important model as its versions serve for defining a number of important complexity classes.

5.1 Nondeterministic Turing machines: Definition

The most straightforward formulation of what is a nondeterministic Turing machine simply adds nondeterminism to a DTM.

Definition 5.1 (NTM, version 1)**.** *The definition of a nondeterministic Turing machine (NTM) differs from the definition of DTM only in that the transition function δ is a multivalued function:*

$$\delta \;:\; Q \times \Gamma^k \to 2^{Q \times \Gamma^k \times \{\leftarrow, \rightarrow, \cdot\}^k},$$

that is, $\delta(q, \vec{c})$ is a non-empty set of triples $(q', \vec{c'}, \vec{d'})$ (and not just a single triple).

*In simple words: an NTM has **one or more instructions** $(q, \vec{c}) \mapsto (\ldots)$ **for specific** q **and** $\vec{c}$.*

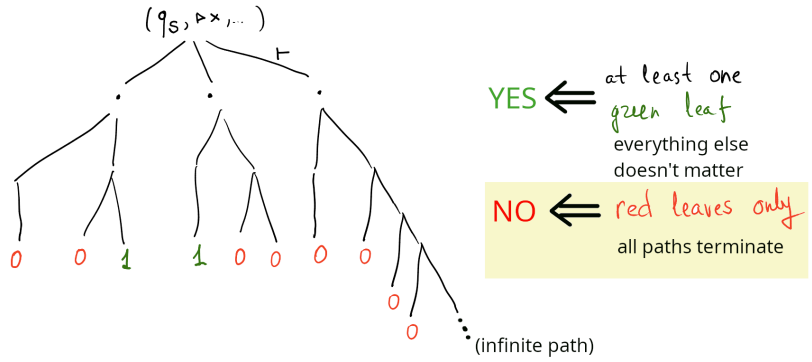
Technical note: For compatibility with other sources we included the possibility that $\delta(\ldots) = \emptyset$. In this case we assume that the machine enters an infinite loop: it continues to make steps, but nothing changes. (It also makes the definition more similar to NFAs.)

Definition 5.2 (A run of an NTM)**.** *A nondeterministic machine makes a nondeterministic choice when it selects a specific instruction out of a set of matching ones.*

So the notions of configuration, $\vdash$, $\vdash^*$ stay the same as for DTMs, but instead of a single triple returned by δ, we can use any member of the set returned by it.

5.2 Computation trees

A computation tree (or a tree of computations) is a very important notion in understanding nondeterministic computations. 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 out from a node.

A computation path is a path from the root to a leaf (or a path starting in the root and not finishing — it's infinite). 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.
There may be also infinite paths — that simply do not end.

5.3 The running time and the result of a NTM

A run of NTM has many possible variants of continuing and finishing (or not finishing). So what is the running time of a NTM and what can we treat as the result of a NTM N on input x?

Definition 5.3 (Running time of NTM)**.** *Consider a specific execution of NTM N, that is, a sequence $(q_S, \triangleright x, 0) \vdash \ldots \vdash (q_Y, \ldots, \ldots)$ or $(q_S, \triangleright x, 0) \vdash \ldots \vdash (q_N, \ldots, \ldots)$ (to avoid ambiguity with different definitions of NTM, let us say that q_Y or q_N is the first stopping state in this sequence).*

The running time of an NTM N on input x, denoted $\text{time}_N(x)$ is the maximum possible number of steps ($\vdash$) in such a sequence from the starting to an accepting or a rejecting configuration. One can also say that it is the depth of the computation tree.

Note that the running time may be infinite even if $N(x)$ has an accepting path.

We will only talk about NTMs that say "yes" or "no" (not functions defined by NTMs). Since the result of an NTM depends on its nondeterministic choices, what does it mean to say "yes" or "no"?

A NTM "accepts" its input iff it *can* reach the accepting state!

For a machine N and an input x, we write:

1. $N(x) = 1$ or $N(x) =$ "yes" iff $(q_S, \triangleright x, 0) \vdash^* (q_Y, \ldots, \ldots)$ (N accepts x).

   Note that there must be at least one accepting computation path (finishing in q_Y), but there may be not only rejecting but also infinite paths.

2. $N(x) = 0$ or $N(x) =$ "no" iff $\text{time}_N(x) < \infty$ and $(q_S, \triangleright x, 0) \not\vdash^* (q_N, \ldots, \ldots)$ (N rejects x).

   That is: all the paths are finite and end in the rejecting state q_N.

3. $N(x) = \nearrow$ otherwise, that is, for every computation path either $N(x)$ finish in the rejecting state q_N or the path is infinite.

Remark 5.1 (NTMs are not symmetric!)**.** *If a DTM stops in finite time, we can invert its answer: replace q_Y with q_N, and vice versa.*

However, we cannot invert the answer of NTM this way even if it stops (all branches are finite). Indeed, if both accepting and rejecting branches existed (a typical situation when $N(x) = 1$), if we simply exchange the final states, there will be still accepting and rejecting branches, so still $N(x) = 1$.

If some branches are infinite, it is even worse: how can we "invert" an infinite branch? This is impossible even for a DTM, we will talk about it soon (Remark 6.1).

6 Decision and recognition

A class of languages is a set of languages, that is, is a subset of 2^{Σ^*}. It is usually convenient to talk about languages over the binary alphabet $\{0, 1\}$ — if what follows, when we talk about languages in a different alphabet, we are assuming a certain binary encoding of this alphabet. Thus, for as a class of languages is a subset of $2^{\{0,1\}^*}$ (the set of all sets of bit strings of finite length).

For example, you surely heard about the complexity classes **P** and **NP** in the algorithms course.

We are going to define rigorously several such classes, but before that we will give two versions of solving a decision problem.

6.1 Decision and recognition, recursive and recursively enumerable languages

Consider a decision problem, i.e., a language $L \subseteq \{0, 1\}^*$. A Turing machine can either just identify (accept) its elements ($x \in L$) or fully decide it, i.e., explicitly reject every $x \notin L$ (in a finite amount of time!).

Indeed, what if a machine is sometimes unable to stop and to give its answer? Consider, for example, the problem of checking whether a machine M given to us (as a bit string) will ever stop (even on the empty input ε)? We can simulate its run using UTM, however, if $M(\varepsilon)$ does

not stop, we won't return the answer in a finite amount of time (whereas if it does stop, we can accept after it comes to q_Y or q_N).

This discussion gives raise to two different notions of solving a decision problem.

Definition 6.1. *A DTM M decides a language L over an alphabet Σ if for every input $x \in \Sigma^*$*

- *If $x \in L$, then $M(x) = 1$,*
- *If $x \notin L$, then $M(x) = 0$.*

In particular, it means that M stops on every input.

If a language is decided by some DTM, we call it decidable or Turing-decidable, or recursive.

Definition 6.2. *A DTM M recognizes (or accepts) a language L over an alphabet Σ if, for every input $x \in \Sigma^*$,*

- *If $x \in L$, then $M(x) = 1$,*
- *If $x \notin L$, then $M(x) \neq 1$ (that is, $M(x) = 0$ or $M(x) = \nearrow$).*

If a language is decided by some DTM, we call it recognizable or Turing-recognizable, or recursively enumerable.

Obviously, if a machine decides a language, it also recognizes it (but not necessarily vice versa!).

Remark 6.1 (Recursive enumerability is not symmetric!)**.** *We talked about asymmetry of NTMs (Remark 5.1.) The same thing happens w.r.t. recognizability and recursively enumerable sets: the notion "decidable" is symmetric while the notion "recognizable" is not.*

Indeed, if L is decidable by a DMT M, we can build a new machine deciding $\overline{L}$ by simply exchanging the states q_Y and q_N of M.

However, if L is only recognizable by M, it does not work. Those inputs that were accepted will be rejected, that's OK. However, we need to accept all the inputs that were not accepted by M, in particular, those that cause M to run infinitely. How can we figure it out that M would not stop, and then stop it and accept? There may be no way to do it.

One can define the notion "M recognizes L" even if M is a NTM — no changes are needed in the definition. Thus for $x \in L$ such an NTM will have at least one accepting path (other paths are either rejecting or infinite), and for $x \notin L$ it will not have such a path (thus all the paths will be either rejecting or infinite). It turns out that for recognizing it does not matter whether we consider deterministic or nondeterministic machines.

6.2 DTMs and NTMs are equivalent wrt recursive enumerability

Theorem 6.1. *For every language $L \subseteq \{0,1\}^*$ there exists a DTM recognizing L if and only if there exists a NTM recognizing L.*

Proof. We need to prove it in one direction only. Assume that N is a NTM recognizing L. We will build a DTM (that is, a deterministic algorithm) doing this job.

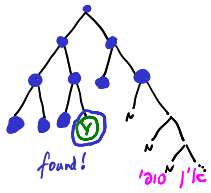
The computation tree of N (on a specific input x) may be infinite, but it can be described in a very constructive way. Namely, we know the configuration in its root (the starting configuration) and we know what are the edges. In particular, we can enumerate all the edges (next possible configurations) exiting a node of this true.

We need to find an accepting path in this tree if it exists (this is the only difference between the cases $N(x) = 1$ and $N(x) \neq 1$: the presence of at least one accepting path). If we tried to do a depth-first search in it, we could well go into an infinite branch. Therefore, we will do a depth-first search.

Starting from the root, our algorithm will write down the list of all the configurations in the next level of this tree (that is, reachable from the starting configuration in a single step). Then it will take the first configuration C in the list and write down additionally all the configurations that can follow C. It will repeat this step until it sees q_Y in one of the configuration — then it will accept.

If $x \in L$, that is, if any accepting path is present in the tree, our algorithm will find its terminal node sooner or later (to see it, notice that if it is locate at depth d and the maximum size of a set $\delta_N(q,a)$ is m, then this accepting configuration will be found after considering at most $1 + m + m^2 + \ldots + d^{m-1} = O(d^m)$ configurations.

If $x \notin L$, then there is no accepting path, and, of course, we will find nothing and will not accept. This is correct. However, we can spend an infinite amount of time while searching for it. This is also OK, because we were trying just to recognize L.



□

6.3 Enumerators

What does the word "enumerable" in "recursively enumerable" really mean? There is a reason for it!

Definition 6.3. *An enumerator for L is an algorithm that lists all members of L.*
(Every $x \in L$ will be printed — sooner or later — and no other word will be printed.)

Remark 6.2. *Note that if our enumerator prints some words more than once, we can easily force it to do it just once:*

- *keep the list of already printed words, and*
- *check a new word against it — if found, don't print it again.*

Theorem 6.2. *L has an enumerator if and only if L is recursively enumerable (i.e., there is a DTM recognizing L).*

Proof. $\Rightarrow$: If we have an enumerator for L, then we can easily recognize L: On input x, run the enumerator for L and check every word it prints — is this x? Once we have found x, accept.

$\Leftarrow$: Given a DTM A that recognizes L, let us build an enumerator for it.

The first idea is to run A in parallel on all possible inputs and print a new string once A accepts something. We do not have any parallel processors, so we will interleave steps of our simulation of A on different inputs so that every "process" will take some of our "processor time", that is, it will not be stuck and will advance from time to time.

It would force us to keep the memory (the configuration) of each of these "parallel processes". To simplify the implementation of this idea let us bound every process to run for some specific number of steps and restart it completely (with a new bound) when we want it to run for longer time.

In other words, we would like to make two cycles:

for $j := 1$ to ∞ do *// Try different time bounds*
  for $m := \epsilon, 0, 1, 00, 01, 10, 11, 100, \ldots$ all the words in the length-increasing lexicographic order
    run $A(m)$ for j steps, if it accepts, then accept.

If we would write it in exactly this way, we would never get to $j = 2$, because the cycle on m is also infinite. To avoid this problem we will enumerate pairs (m, j) in a way that will enable to come to each pair in a finite amount of time. The construction is similar to a popular proof of $|\mathbb{Q}| = |\mathbb{Z}|$.

Let us identify every input $m = b_1 b_2 \ldots b_n \in \{0,1\}^*$ with the natural number $\underbrace{1\, b_1 \ldots b_n}_{\text{binary encoding}}$.

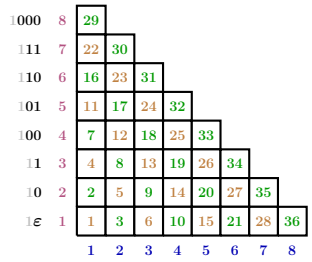
We give a distinct number $u(m, j)$ to every pair $(m, j) \in \mathbb{N} \times \mathbb{N}$ like described in the picture (the numbers in the triangle are the values of u, and the coordinates are j and m).

Our algorithm works in stages. During the stage number $u(m, j)$ our algorithm simulates j steps of A on input m. If A gets to q_Y at this step, the algorithms prints m.

Let us illustrate the stages and the order of executions:

1. Run $A(\epsilon)$ for 1 step. If it accepts, print ϵ.
2. Run $A(0)$ for 1 step. If it accepts, print 0.
3. Run $A(\epsilon)$ for 2 steps. If it accepts, print ϵ.
4. Run $A(1)$ for 1 step. If it accepts, print 1.
5. Run $A(0)$ for 2 steps. If it accepts, print 0.
6. Run $A(\epsilon)$ for 3 steps. If it accepts, print ϵ.
7. Run $A(00)$ for 1 step. If it accepts, print 00.

...



This way each element of L will be printed infinitely many times (but we have already noticed in Remark 6.2 that we can force the algorithm to print only one sample of every element). □

7 Classes of languages (decision problems)

In this section we assume that all the inputs are encoded in the binary alphabet (as we know, we can always re-encode the inputs).

A class $\mathcal{C}$ of languages is a set of languages. That is, $\mathcal{C} \subseteq 2^{\{0,1\}^*}$.

Let us define several classes of languages without time restrictions (computability classes) and then classes with time restriction (complexity classes).

7.1 Classes for unbounded-time computation: R and RE

Recall that recursive languages are those that are decidable by DTMs.

Definition 7.1 (recursive languages)**.** *The class of languages* $\mathbf{R}$ *consists of all recursive languages (i.e., languages over the alphabet* $\{0,1\}$ *that are decidable by DTMs). That is,* $L \in \mathbf{R}$ *if and only if there exists a DTM* M *such that* M *decides* L*.*

The word "recursive" stems from "recursive functions", a notion in mathematical logic related to the Church–Turing thesis.

Recall that recursively enumerable languages are those that DTMs can recognize.

Definition 7.2 (recursively enumerable languages)**.** *The class of languages* $\mathbf{RE}$ *consists of all recursively enumerable languages (i.e., languages over the alphabet* $\{0,1\}$ *that are recognizable by DTMs.) That is,* $L \in \mathbf{RE}$ *if and only if there exists a DTM* M *such that* M *recognizes* L*.*

We have already seen that these are languages that have enumerators.

Trivially, $\mathbf{R} \subseteq \mathbf{RE}$. Let us prove that these classes are actually different.

Theorem 7.1. $\mathbf{R} \neq \mathbf{RE}$ *(in particular,* $\mathtt{AP} \in \mathbf{RE} \setminus \mathbf{R}$*).*

Proof. We will prove that the following language is in $\mathbf{RE} \setminus \mathbf{R}$:

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

This language (the acceptance problem) asks to tell whether a given M accepts a given input x.

It is easy to see that $\mathtt{AP}$ is recursively enumerable ($\mathtt{AP} \in \mathbf{RE}$). Here is a DTM that recognizes it:

- Run UTM($\langle M, x \rangle$).

Let us prove that $\mathtt{AP} \notin \mathbf{R}$. Assume that there is a DTM H that decides it, i.e.,

$$M(x) = 1 \iff \langle M, x\rangle \in \mathtt{AP} \iff H(\langle M, x\rangle) = 1$$

and

$$M(x) \neq 1 \iff \langle M, x\rangle \notin \mathtt{AP} \iff H(\langle M, x\rangle) = 0.$$

In particular, H stops on every input.

We are going to build a new DTM D that always stops and satisfies two conditions:

$$\begin{aligned} &\text{if } M(\langle M\rangle) = 1, \text{ then} \quad D(\langle M\rangle) = 0 \\ &\text{if } M(\langle M\rangle) = 0, \text{ then} \quad D(\langle M\rangle) = 1 \end{aligned}$$

How can we build it?

On input $\langle M\rangle$, our machine D will run $H(\langle M, \langle M\rangle\rangle)$ with q_Y and q_N exchanged. Therefore, D will always stop (because H does) and return the negation of H. Then our conditions are satisfied.

Let us check all possible cases: what is the value of $D(\langle D\rangle)$? What is the result of applying D to its own description? Our conditions yield that

- If $D(\langle D\rangle) = 0$, then $D(\langle D\rangle) = 1$. Contradiction!
- If $D(\langle D\rangle) = 1$, then $D(\langle D\rangle) = 0$. Contradiction!

Note that these are all possible cases, because D always stops. Since these two cases are impossible as well, it means that D cannot exist, and our assumption on the existence of H was wrong. □

Uniform vs non-uniform models. Recall that a uniform model of computation uses a single finite device for all possible inputs. Therefore, DTMs and NTMs are uniform models of computation. We can now show that non-uniform models of computations (namely, families of Boolean circuits) can very easily decide problems that uniform models cannot decide at all. (Recall that by Church–Turing thesis we can just consider DTMs as an uniform model.)

First of all, let us define a more convenient version of the acceptance problem. It asks whether the given DTM stops (even without any input); the DTM's Turing number is given in unary.

$$\boxed{1^n \in \mathrm{UAP}_\varepsilon \iff T_n(\epsilon) \text{ accepts}}$$

Here n is a Turing number, T_n is the n-th DTM.

This version is also undecidable — let us prove it!

Proposition 7.1. UAP_ε *is undecidable.*

Proof. Assume that $\mathrm{UAP}_\varepsilon \in \mathbf{R}$, thus there exists a DTM D deciding it. Let us build a DTM deciding AP (we know it's impossible!). On input $\langle M, x\rangle$, this DTM will hardwire M and x into the description of a new machine Q and run D on the Turing number of Q (written in unary).

Input (for AP): $\langle M, x\rangle$.
Algorithm:

- Write the decription of machine Q (both M and x are hardwired into it — it's a big machine!)

  ```
  DMT Q:                                          // It does not use its input tape
  Write x on the input tape.
  Simulate M(x).
  Whenever M(x) accepts, accept.
  Otherwise if M(x) rejects, cycle forever.
  ```

- Not just write it! Write it in unary! As $\boxed{1^q}$!
  (Treat $|\langle Q\rangle|$ as a natural number — Turing number — and write this number q of 1's on the tape.)

- Simulate $D(1^q)$.

Since D decides UAP_ε, it always stops, so in particular it stops on 1^q. $D(1^q)$ accepts $\iff$ $1^q \in \mathrm{UAP}_\varepsilon$ $\iff$ Q accepts ε $\iff$ $M(x)$ accepts. Therefore, D indeed decides AP, which is impossible by Theorem 7.1. □

Now observe that there is a very "easy" family of Boolean circuits that decides AP.
If $1^n \in \mathrm{UAP}_\varepsilon$ (that is, $T_n(\epsilon)$ halts), then let $C_n(x) = \bigwedge_{i=1}^n x_i$ (simply check it's $11\ldots1$).
If $1^n \notin \mathrm{UAP}_\varepsilon$, then let $C_n(x) = 0$ (a constant circuit!).
Note that we don't need to do anything meaningful here: there *exists* a designated circuit for each input length, we need just 1 bit of information per input length to build it. However, we do *not* build C_n by an algorithm, we just know this C_n exists!

This family of Boolean circuits $\{C_n\}_n$ solves an undecidable problem! Yet ... we don't know how to construct it algorithmically!

Remark 7.1. *Circuit families (of much larger size) can solve absolutely all problems. Indeed, split language L into parts $L_n = \{0,1\}^n \cap L$. The circuit for n input bits must compute the Boolean function*

$$C_n(x_1, x_2, \ldots, x_n) = \begin{cases} 1, & x \in L \\ 0, & x \notin L \end{cases}$$

Every Boolean function $f : \{0,1\}^n \to \{0,1\}$ can be computed as

$$f(x_1, x_2, \ldots, x_n) = \bigvee_{d \in L} x_i \oplus d_i \oplus 1$$

(observe that in terms of the circuit d_i here are constants, so $\oplus$ is actually either the negation or the identity).

An alternative definition of RE. We can give an equivalent definition of **RE** using the existential quantifier.

Theorem 7.2. *$L \in \mathbf{RE}$ if and only if there exists a language $S \in \mathbf{R}$ such that*

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

Proof. $\boxed{\Leftarrow}$ Assume that such a language S exists, and consider a DTM M_S that decides it. The following machine recognizes L: for every $w \in \{0,1\}^*$ (in length-increasing and then lexicographic order) run $M_S(\langle x, w\rangle)$, and if it accepts — then accept.

Indeed, if $\exists w \; \langle x, w\rangle \in S$, then this w will be found (note that M always stops). If it does not exist, it will not be found, and the machine will never stop.

$\boxed{\Rightarrow}$. Assume that $L \in \mathbf{RE}$, thus there is a DTM M that recognizes L. If it accepts x, then it accepts it in some specific number of steps — we can try all the possibilities. Define

$$S = \{\langle x, t\rangle \;\; : \;\; x \in \{0,1\}^*, \;\; t \in \mathbb{N}, \;\; M(x) = 1, \;\; \mathrm{time}_M(x) \leqslant t\}.$$

As we just said, $x \in L \iff \exists t \in \mathbb{N} \; \langle x, w\rangle \in S$ (we can identify bit strings with natural numbers in any reasonable way).

Let us prove that $S \in \mathbf{R}$. Indeed, on input $\langle x, t\rangle$ let us run M with an "alarm clock" t: if M accepts within t steps, we accept; otherwise we reject. This machine always stops and conforms to the definition of S. □

7.2 Classes for bounded-time computation

7.2.1 Classes DTime and P

What is the running time of a machine M on a certain input x? For RAMs there are several notions of the running time (not discussed in this course). However, for DTMs this is unequivocal.

Recall that we defined $\mathrm{time}_M(x)$ as the number of steps that M performs on input x before reaching the final state (q_Y or q_N). Note that $\mathrm{time}_M(x)$ is a function $\{0,1\}^* \to \mathbb{N}$. We can also define the worst-case running time of M as

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

that is, the maximum of the running time that M takes on inputs of size n. This 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 $O(\ldots)$ or even up to a polynomial).

Definition 7.3 (worst-case time bound)**.** *Let $f : \mathbb{N} \to \mathbb{N}$ be a function or a class of functions (e.g., $O(n^2)$). Consider a DTM M that decides a language L.*

If $\forall n \in \mathbb{N}$ wc-time$_M(n) \leqslant f(n)$ (in particular, for every possible input $x \in \{0,1\}^*$, time$_M(x) \leqslant f(|x|)$), then we say that M decides[1] L in time $f(n)$, where $n = |x|$ is the length of the input.

[1]Note that when there is some upper bound on the worst-case running time of a DTM, there is no difference between "decide"/"accept"/"recognize".

One can define similarly the notion "M computes g in time f", if M computes a function (namely, g).

Let us define the classes **DTime** of decision problems that can be solved within a certain time bound:

Definition 7.4. *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 $f(n)$.*

Remark 7.2. *It is natural to conjecture that* **DTime** *with a larger time bound f is a strictly larger class than* **DTime** *with a smaller time bound f'. This is indeed so (for "nice" functions f, f'), and we can even prove it if f and f' are far enough from each other — we won't prove it in this course (the proof is not difficult, but a bit technical; it uses the diagonalization method and a UTM efficient enough).*

Typically, we talk about $\mathbf{DTime}[O(f(n))] = \bigcup_{g=O(f)} \mathbf{DTime}[g(n)]$.

The class of polynomial-time decidable languages is called **P**. We define it formally as

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

We can define also the class of exponential-time decidable languages as

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

Since all languages in the **DTime** and **P** (and even **EXP**!) classes can be decided (and we even know an upper bound on the running time of the deciding machine), trivially $\mathbf{P} \subseteq \mathbf{EXP} \subseteq \mathbf{R}$ and $\mathbf{DTime}[f(n)] \subseteq \mathbf{R}$.

Example 7.1 (regular languages)**.** *The class of regular languages in the binary alphabet,* `REG`*, is a subset of* $\mathbf{DTime}[O(n)]$*. To see this, observe that in order to decide a language that is accepted by a deterministic finite automaton, it suffices to simulate this automaton on the given input. We are doing it for a specific regular language, that is, for a fixed automaton. The simulating DTM will move the head to the first symbol of the input and then simply work as the automaton with the only difference: accepting states of the automaton are not accepting states of the DTM. Instead, when our DTM comes to the end of the input (that is, it looks at the "blank space" character) it will accept if the automaton is in an accepting state and it will reject otherwise. Since we are talking about a DFA, this simulation will take just $n + 1$ steps: at each step either $\triangleright$ or the next symbol of the input is read.*

7.2.2 Classes NTime and NP

We have already define the time taken by a nondeterministic machine N on an input x as the maximum possible running time over all its computational paths.

Definition 7.5 (**NTime**, **NP**). *We say that L belongs to the class* $\mathbf{NTime}[f(n)]$ *if there is an NTM N such that*

- *N decides L,*
- *$\forall n \in \mathbb{N}$ wc-time$_N(n) \leqslant f(n)$. (Recall that for NTM the running time is the maximum running time over all possible nondeterministic choices.)*

We define also $\mathbf{NP} = \bigcup_{k \in \mathbb{N}} \mathbf{NTime}[O(n^k)]$.

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

Example 7.2 (SAT). *Satisfiable Boolean formulas in CNF:*

$$\texttt{SAT} = \{F \text{ in CNF of } n \text{ variables } x_1, \ldots, x_n : \exists x_1, x_2, \ldots, x_n \in \{0,1\}^n \text{ such that } F(x_1, x_2, \ldots, x_n) = 1\}.$$

An NTM that decides this language simply guesses nondeterministically the values for $x_1, x_2, \ldots, x_n$ and then evaluates F for them.

We will give two more definitions for NTM (thus, in total, we will have three definitions for the class **NP**) and prove the equivalence of all of them.

NTM and NP, version 2.

Definition 7.6 (NTM, version 2). *A NTM is a DTM with a distinguished "witness" tape. This "witness" (a finite string) is written on it in the starting configuration.*

Notation: to denote the result of this machine, we write $M(x, w)$, where x is the input and w is the witness.

Since the machine is now deterministic, we know well how it behaves. However, what do we mean by its answer on x, "yes" or "no", if it can be given different witnesses?

Definition 7.7 (NTM (version 2), semantics). *A NTM M (version 2) decides a language L if for every input x,*

- *if $x \in L$, then $\exists w \in \{0,1\}^*$ such that $M(x, w)$ accepts.*
- *If $x \notin L$, then $\forall w \in \{0,1\}^*$, $M(x, w)$ rejects.*

Don't get shocked by the apparently unbounded length of w: of course, a machine cannot read more cells than the number of steps it makes before the stop. We know well what is the running time of $M(x, w)$ as a DTM. However, we do not take a specific w into account when we define $\text{time}_M(x) = \max_{w \in \{0,1\}^*} M(x, w)$ (this is the running time of M "as a nondeterministic machine, version 2"). Thus a polynomial-time NTM of this version must stop after at most $p(n)$ steps, where p is a polynomial and $n = |x|$ (no mention of w here!).

Naturally, the second definition of **NP** sets it to the class of languages decided by polynomial-time NTMs of version 2.

Let us prove the equivalence of the two definitions.

Theorem 7.3. *L belongs to the version 1 of* **NP** $\iff$ *L belongs to the version 2 of* **NP**

Proof. We show how to transform a machine of one type into an equivalent machine of another type. It will be easy to see that (1) they are equivalent, that is, the existence of an accepting path for NTM (version 1) corresponds to the existence of a witness for NTM (version 2), (2) they stop in roughly the same number of steps (that is, the overhead is upper bounded by a polynomial in the length of the input).

Version 1 $\Rightarrow$ Version 2. First of all, note that we can assume w.l.o.g. that version-1 NTM always chooses between two configurations only. In order to see this, transform its transition function δ into δ' such that $|\delta'(q,c)| \leqslant 2$. One can use additional states to distinguish between several possibilities: for example, if there are four elements r_0, r_1, r_2, r_3 in $\delta(q,c)$, we can introduce two new states q_0 and q_1 and replace the four transitions $(q,c) \mapsto r_i$ (where $i \in \{0,1,2,3\}$) by the transitions

$$\begin{aligned}(q,c) &\mapsto (q_0,\ldots),\\ (q,c) &\mapsto (q_1,\ldots)\end{aligned}$$

(not modifying the symbols or the head positions) and the transitions

$$\begin{aligned}(q_0,c) &\mapsto r_0,\\ (q_0,c) &\mapsto r_1,\\ (q_1,c) &\mapsto r_2,\\ (q_1,c) &\mapsto r_3.\end{aligned}$$

We now transform such an NTM N into an equivalent version-2 NTM M. While M cannot do nondeterministic steps, it has a witness. It will use it to simulate the nondeterminism. Namely, imagine that N has to choose between two instructions

$$\begin{aligned}(q,c) &\mapsto (q',c',d'),\\ (q,c) &\mapsto (q'',c'',d'').\end{aligned}$$

Then M will read the next character from the witness tape[2] (and send the head to the right) and behave accordingly:

$$\begin{aligned}(q,c0) &\mapsto (q',c'0,d' \to),\\ (q,c1) &\mapsto (q'',c''1,d'' \to).\end{aligned}$$

That is, if the next bit of the witness is 0, M will go to q', and if it is 1, it will go to q''.

It is easy to see that the steps of M (in particular, their number) correspond to the steps of N, and a run of M for a witness w of appropriate length corresponds to one of the computational paths of N (if the length is inappropriate, it may be even smaller).

[2]Recall that $(q,c0)$ means that the symbol c is read on the first tape, and the symbol 0 is read on the second tape (which is now the witness tape).

Version 2 ⇒ Version 1. To transform a version-2 NTM M into a version-1 NTM N, devote one empty tape of N to w. We do not have this witness yet, but we can generate it ourselves nondeterministically and then run the original version-2 NTM.

Since we know the running time bound (we do know it if it's a polynomial), our machine first computes it and then simply writes a witness of at most this length nondeterministically. (We write 0's and 1's nondeterministically, and also stop doing that nondeterministically.) After that, it proceeds like M. An important thing here is that once w is written, it can be used many times (this is important, because M assumes that it gets a specific w, and its bits are not regenerated nondeterministically each time they are accessed).

The only addition to the running time is when we write the witness (of length bounded by a known polynomial in the input length!). □

Remark 7.3. *Alternatively, we can generate the witness on demand: write the next bit w_i nondeterministically when we need it. (Technical details are an easy exercise though there are many technicalities on this road.)*

Connection between the nondeterministic and deterministic time complexity in the time-bounded case. We learned that in the unbounded-time setting (recognizability, not decidability) DTMs and NTMs accept the same class of languages. What about the time-bounded setting?

Trivially, $\mathbf{DTime}[f(n)] \subseteq \mathbf{NTime}[f(n)]$ and $\mathbf{P} \subseteq \mathbf{NP}$. What about the other direction?

It turns out that again it is possible to build a deterministic machine equivalent to a nondeterministic one, yet for an exponential price. (We formulate this theorem for polynomial time, but we could prove it also for any other reasonable time bound.)

Theorem 7.4. $\mathbf{NP} \subseteq \mathbf{EXP}$.

Proof. Let $L \in \mathbf{NP}$. Consider a version-2 NTM M deciding L in time $p(n)$, where p is a polynomial. Our deterministic machine works as follows:

1. Given input x, compute $m := p(|x|)$.
2. For every $w \in \{0,1\}^*$ of length at most m, simulate the work of $M(x,w)$.

Obviously, this machine decides L and its running time is $2^{p(|x|)}$ multiplied by time needed to simulate $M(x,w)$, which is also bounded by a polynomial. The first step also takes polynomial time. Therefore, the total running time is bounded by an exponent of a polynomial of $|x|$. □

In total, we know that

$$\texttt{REG} \subseteq \mathbf{P} \subseteq \mathbf{NP} \subseteq \mathbf{EXP} \subseteq \mathbf{R} \subseteq \mathbf{RE}$$

7.3 The complement

Reminder / Warning: When we are talking about a language, it is important to fix the alphabet Σ that we are working with: $\overline{L} = \Sigma^* \setminus L$. In particular, even if L is unary, it is important to know whether we are considering languages over the unary or the binary alphabet (or over a different alphabet).

We have already seen that classes **RE** and **NP** are not symmetric: if L belongs to such a class, there is no clear way to build a machine of the same type for $\overline{L}$. This gives rise to new classes: classes of complements.

Definition 7.8 (co-)**.** *For a class $\mathcal{C}$, denote*

$$\textbf{co-}\mathcal{C} = \{L \,:\, \overline{L} \in \mathcal{C}\}.$$

Note that $\texttt{REG} = \textbf{co-}\texttt{REG}$ (regular languages are closed under the complement), $\mathbf{R} = \textbf{co-R}$, $\mathbf{EXP} = \textbf{co-EXP}$, $\mathbf{P} = \textbf{co-P}$ (because we can simply exchange the states q_Y and q_N, the running time does not change, but the answer gets inverted). The situation with **RE** and **NP** is different: their notion of "yes" and "no" depends on quantifiers, and the existential quantifier $\exists$ turns into the universal quantifier $\forall$ if we consider the complement of a language (the negation of a condition).

Example 7.3 (Effective reasonable encoding for SAT)**.**
$\texttt{SAT} = \{F \text{ in CNF} : n \geqslant 0,\ F \text{ contains } n \text{ variables},\ \exists x_1, \ldots, x_n \in \{0,1\}\ F(x_1, \ldots, x_n) = \texttt{True}\}$.

We know that $\texttt{SAT} \in \mathbf{NP}$ *(a good witness is a string of "good" values for $x_1, \ldots, x_n$, called a satisfying assignment). Formulas belonging to* SAT *are called satisfiable formulas.*

Then $\overline{\texttt{SAT}} \in \textbf{co-NP}$*. However, what is* $\overline{\texttt{SAT}}$*?*

In order to talk absolutely formally about SAT *and* $\overline{\texttt{SAT}}$ *we need to define a specific reasonable encoding of CNFs! Then it is useful to define it so that incorrectly formed strings correspond to a specific trivial formula (say, $x_1 \wedge \overline{x_1}$) by definition. With such an encoding,*
$\overline{\texttt{SAT}} = \{F \text{ in CNF} : n \geqslant 0,\ F \text{ contains } n \text{ variables},\ \forall x_1, \ldots, x_n \in \{0,1\}\ F(x_1, \ldots, x_n) = \texttt{False}\}$

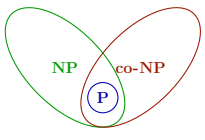
Formulas belonging to SAT *are called unsatisfiable formulas.*

Exercise 7.1 (propositional tautologies)**.** *Show that the following language belongs to* **co-NP***:*
$\texttt{TAUT} = \{F \text{ in DNF} : n \geqslant 0,\ F \text{ contains } n \text{ variables},\ \forall x_1, \ldots, x_n \in \{0,1\}\ F(x_1, \ldots, x_n) = \texttt{True}\}$

Remark 7.4. *When solving Ex. 7.1, observe that the negation of CNF is a DNF. Consider generalizations of* SAT *and* TAUT*: use Boolean circuits instead of CNFs and DNFs. Observe that the negation of a Boolean circuit is still a Boolean circuit!*

7.4 General picture and major open questions

Contrary to the situation with $\mathbf{R} = \mathbf{RE} \cap \textbf{co-RE}$, it is unknown whether there are languages in $\mathbf{NP} \cap \textbf{co-NP}$ that cannot be decided in polynomial time:



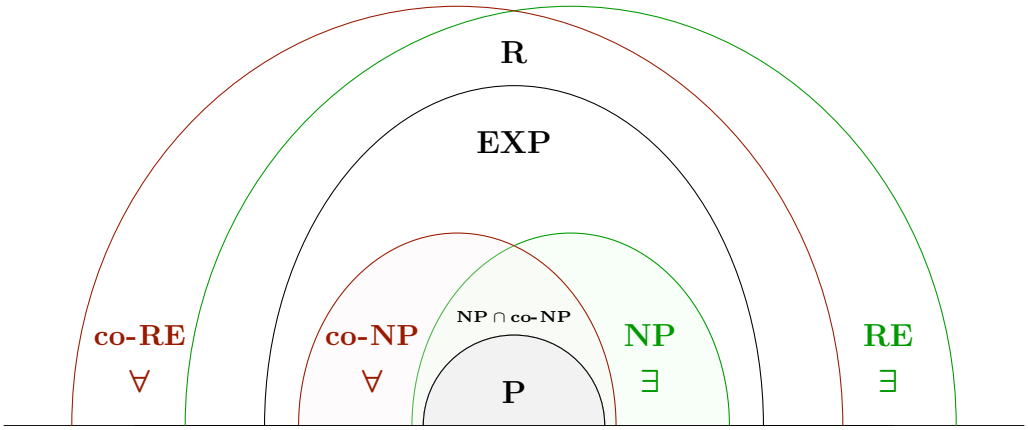


Figure 2: Computability and complexity classes

So we have the following major questions, all of them are still open

1. **P** = **NP**?
2. **NP** = **co-NP**?
3. **P** = **NP** ∩ **co-NP**?

Obviously, solving the first question positively would solve all of them positively. However, even solving the second one positively does not answer the third question. Moreover, it is known (though we do not prove it in this course) that **P** ≠ **EXP**. Therefore, either **P** ≠ **NP** or **NP** ≠ **EXP** (or both), but we still cannot prove any of these non-equalities!

8 P.S. Search problems

In the most part of this course we considered only yes/no problems (also called decision problems, languages, predicates). Now it's time to consider also search problems: indeed, the definition of

the class **NP** suggests that one may want to search for a witness in order to make sure that the input is in the language.

Before defining them, let us give a yet another definition of **NP**.

8.1 NP, version 3

We now give the third definition of **NP** that may look more like what you have seen in the algorithms course.

Recall that a binary relation R on S is a set of pairs, or a subset of $S \times S$. We say that $R(x, w)$ is true if $(x, w) \in R$; it is false if $(x, w) \notin R$. We consider relations on $\{0,1\}^*$.

For a pair (x, w), we call $\boldsymbol{x}$ (an **instance** of) the problem R and we call w a candidate solution.

Example 8.1. *For the problem* `COMPOSITENESS` *stated as finding a non-trivial divisor, the relation would be*[3]

$$R(x, w) = \{(x, w) \,:\, w, x \in \mathbb{N},\ w|x,\ w \neq 1, x\}.$$

Then the composite number $x = 39$ is an instance of this problem, and $w = 13$ is a correct solution to it, while $w = 14$ is an incorrect solution. The instance $x = 17$, being a prime number, has no correct solutions at all.

Given a binary relation, one can think of a search problem and a decision problem associated with it.

Definition 8.1 (search problem)**.**
Given a binary relation R, the search problem associated with it is:

Given x, find w such that $R(x, w)$ is true (or say "no" if there is no such w).

Definition 8.2 (decision problem associated with a relation)**.**
Given a binary relation R, the decision problem associated with it is:

Given x, say whether there exists w such that $R(x, w)$ is true.

In symbols, the language is $L = \{x \in \{0,1\}^* \,:\, \exists w \in \{0,1\}^*\ R(x, w)\}$.

Definition 8.3. *A relation $R \subseteq \{0,1\}^* \times \{0,1\}^*$ is polynomially bounded if there is a polynomial p such that for every $x \in \{0,1\}^*$, if $\exists w\ R(x, w)$ then*[4] *$\exists w \in \{0,1\}^{\leqslant p(|x|)} R(x, w)$.*

In other words, R is polynomially bounded if once there is a correct solution, then there must be also a "short" (polynomial-size) correct solution.

Definition 8.4. *A relation $R \subseteq \{0,1\}^* \times \{0,1\}^*$ is polynomially verifiable if $R \in \mathbf{P}$.*

In other words, R is polynomially verifiable if one can decide in polynomial time the language consisting of all the pairs (x, w) such that $R(x, w)$ is true.

[3] We identify $\{0,1\}^*$ with $\mathbb{N}$.

[4] Notation: $\Sigma^{\leqslant k}$ denotes the set of all strings of length at most k in the alphabet Σ.

Now the third definition of **NP** is

Definition 8.5. *A language L belongs to the class* **NP** *if there is a polynomially bounded and polynomially verifiable binary relation R such that for every $x \in \{0,1\}^*$,*

- $x \in L \iff \exists w\ R(x,w)$.
- $x \notin L \iff \forall w\ \overline{R}(x,w)$.

Let us show that it is equivalent to the version 2 of **NP**.

If we have a polynomial-time NTM (version 2) M, then consider M' that does the same, but both x and w are on the same (input) tape. Thus on input $\langle (x,w) \rangle$ it starts with moving w to another tape and then proceeds as M. Then M' defines a polynomially verifiable and also polynomially bounded relation (it simply cannot read w of a larger length).

On the other hand, if we have a polynomially verifiable and polynomially bounded relation R, then we have a polynomial-time NTM (version 2) for the corresponding language. The difference is only that it assumes w to be given on the witness tape.

Example 8.2 (**NP** versions for COMPOSITENESS)**.** *Consider once again the language*

$$\texttt{COMPOSITENESS} = \{x \in \mathbb{N} : x \text{ has a nontrivial divisor}\}.$$

NP version 3: *Relation* $R(x,w) = \{(x,w) : x, w \in \mathbb{N},\ w|x,\ w \neq 1, x\}$.

NP version 2: $M(x,w)$ *gets witness* w*, checks that* $w|x,\ w \neq 1, x$.

NP version 1: $N(x)$ *verifies the same as in version 2 without getting* w*: it guesses* w *nondeterministically and then checks that* $w|x,\ w \neq 1, x$*. To generate* w*, it reads* x *and simultaneously writes* w *on another tape:*

$$\begin{aligned}
(q_{guess}, 0_) &\mapsto (q_{guess}, 00, \rightarrow) \text{ generated 0} \\
(q_{guess}, 0_) &\mapsto (q_{guess}, 01, \rightarrow) \text{ generated 1} \\
(q_{guess}, 0_) &\mapsto (q_{check}, 0_, \bullet) \text{ enough!} \\
(q_{guess}, 1_) &\mapsto (q_{guess}, 10, \rightarrow) \text{ generated 0} \\
(q_{guess}, 1_) &\mapsto (q_{guess}, 11, \rightarrow) \text{ generated 1} \\
(q_{guess}, 1_) &\mapsto (q_{check}, 1_, \bullet) \text{ enough!} \\
(q_{guess}, __) &\mapsto (q_{check}, __, \bullet) \text{ no more input, definitely enough!}
\end{aligned}$$

8.2 A version of NP for search problems

The third definition of **NP** naturally gives rise to search problems. Such a problem is associated with a binary relation R; namely, given x output w such that $R(x,w) = \texttt{True}$ (or say "no" if such w does not exist).

If R is polynomially bounded and polynomially verifiable, then we call this problem an **NP**-search problem.

A historical note: Leonid Levin defined **NP** *problems exactly as search problems.*

Given x, find w such that $R(x, w)$ is true (or say "no", if there is no such w).

Let us call **Search-NP** the class of such problems (relations).

Example 8.3 (Examples of problems in **Search-NP** (and not in **Search-NP**))**.**
The following problems are in **Search-NP***:*

- *Find a satisfying assignment for a formula in CNF.*
- *Find a nontrivial divisor for a natural number.*
- *Find an isomorphism of two given graphs.*

The following is not a problem in **Search-NP***, at least, it is unclear:*

Find a strategy for winning the infinite Go game on a $n \times n$ board (starting at a given position).

Why? Let us start with the fact that a strategy is a function, so one needs too many bits to describe it. Then, how can one bound the number of steps in the game? (I even do not mention here that understanding what is the outcome of a particular Go game is nontrivial.)

We are interested in **NP**-search problems R that can be solved in polynomial time, that is, there is a DTM M and a polynomial p such that given x, it finds a correct solution w (that is, stops in q_Y with w written on its output tape such that $R(x, w) =$ `True`) in time $p(|x|)$. *(Note that if there is a time bound for inputs that have solutions, one can interrupt the work of M after $p(|x|)$ steps and conclude that there are no solutions if none are found before that.)*

Example 8.4 (**Search-NP**-problems solvable in polynomial time)**.** *You learned these algorithms in Algorithms-1,2 courses:*

- *Find a topological order for vertices of a DAG.*
- *Find a satisfying assignment for a formula in 2-CNF.*

There may be many search problems for the same decision problem.
Very important!

The same decision problem L may correspond to different search problems.
To check that the input is a composite number, we can use different types of witnesses:

- a nontrivial divisor,
- a nontrivial number $y \neq kx$ such that $gcd(x, y) \neq 1$,
- solutions to various number-theoretical problems.

It may lead to search problems of very different hardness, while the decision problem will stay the same: `COMPOSITENESS`.

Moreover, it is known that the decision problem `COMPOSITENESS` (and some of the search problems associated with it) can be solved in polynomial time, while finding a nontrivial divisor stays a very challenging problem: if we would be able to solve it efficiently, then we would break the RSA function, which is an important cryptographic primitive currently assumed to be secure.

8.3 Levin's optimal algorithm for Search-NP problems

This section is not intended for the exam!

Even if we do not know how to solve SAT efficiently, it may be the case that **P** = **NP**. Can we find satisfying assignments in polynomial time *without knowing* this algorithm?

Levin's result says — yes! Moreover, if we can do it for a subset of instances of some other **Search-NP** problem, then this algorithm will do it as well.

The idea of the algorithm is

- To try all possible algorithms "in parallel".
- Having no access to parallel machines, run this on a single DTM called OPT so that each algorithm gets a constant share of OPT's running time (so it is *slowed down by a constant factor only*, this constant depends on the algorithm).
- Even after accounting for the technicalities of the simulation, a polynomial-time run will still be a polynomial-time run.
- A crucial factor is that we can always verify if an algorithm returned something meaningful: this is the essence of **NP**.

Here is the "code". Recall the enumeration of all DTMs (we are interested in DTMs that compute functions) $T_1, T_2, T_3, \ldots$, called giving "Turing numbers" to the machines. Let R be a polynomially verifiable polynomiall bounded binary relation corresponding to our search problem; let us denote the polynomial-time DTM checking $R(x, w)$ by the same letter.

OPT(x):

for $i = 1, 2, \ldots$ do:
- parse[5] i as $i = 2^\ell(1 + 2k)$, getting the numbers ℓ and k;
- simulate the k-th step of $U(\langle T_\ell, x\rangle)$;
- if it stops at this step, check its output w using $R(x, w)$ and if $R(x, w)$ is true, then print w and stop.

The technicalities left behind are:

- For each "process" number ℓ, namely $U(\langle T_\ell\rangle, x)$, we need to keep its configuration between its steps so that each process uses its own portion of the tape,
- We need to switch between the processes.

The implementation of these technicalities depends on the model of computation, for example, they are more efficient on RAM. However, it is clear that the overhead is polynomial. So if your favorite algorithm solves the problem for a series[6] of inputs $x^{(1)}, x^{(2)}, \ldots$ in polynomial time, then OPT will solve it in polynomial time as well.

Remark 8.1. *We work efficiently enough for x that has a solution w such that $R(x, w)$ is true. We do not stop on inputs that have no solution at all. We never stop on them!*

The End (for Spring 2025)

References

[Sipser] M. Sipser. Introduction to the Theory of Computation.

[Papadimitriou] C. H. Papadimitriou. Computational Complexity.

[5] That is, count the number of 0's at the end of the binary representation of i. This is ℓ. Throw these 0's out, you now have an odd integer number. Drop the last bit of it (it is 1), now you get k.

[6] Why "series"? Because we need an infinite number of inputs for talking about asymptotics, in particular for saying the words "polynomial time". "All satisfiable formulas" are also a "series of inputs".