Introduction to Complexity and Computability

NTIN090

Petr Kučera

2019/20

Introduction

Syllabus

1. Turing machines and their variants, Church-Turing thesis
2. Halting problem and other undecidable problems
3. RAM and their equivalence with Turing machines. Algorithmically computable functions
4. Decidable and partially decidable languages and their properties
5. m-reducibility and m-complete languages
6. Rice's theorem
7. Nondeterministic Turing machines, basic complexity classes, classes P, NP, PSPACE, $\mathrm{EXPTIME}$
8. Savitch's theorem
9. Deterministic space and time hierarchy theorems
10. Polynomial reducibility among problems NP-hardness and NP-completeness
11. Cook-Levin theorem, examples of NP-complete problems, proofs of NP-completeness
12. Pseudopolynomial algorithms and strong NP-completeness
13. Approximations of NP-hard optimization problems, approximation algorithms and schemes
14. Classes $\mathrm{co\text{-}NP}$ and $\#\mathrm{P}$

Literature

Both

- Sipser, M. *Introduction to the Theory of Computation.* Vol. 2. Boston: Thomson Course Technology, 2006.

Computability

- Soare R.I.: *Recursively enumerable sets and degrees.* Springer-Verlag, 1987
- Odifreddi P.: *Classical recursion theory*, North-Holland, 1989

Complexity

- Garey, Johnson: *Computers and intractability — a guide to the theory of NP-completeness*, W.H. Freeman 1978
- Arora S., Barak B.: *Computational Complexity: A Modern Approach*. Cambridge University Press 2009.

Motivational questions

1. What is an algorithm?
2. What can be computed using algorithms?
3. Can all problems be solved using algorithms?
4. How can we recognize whether a given problem can be solved by an algorithm?
5. Which algorithms are “fast” and which problems can be solved with them?
6. What is the difference between time and space?
7. Which problems are “easy” and which are “hard”? How can we recognize them?
8. Is it easier to examine or to be examined?
9. How we can solve problems for which we do not know any “fast” algorithm?

Computability

A Light Introduction into the Theory of Algorithms

The first program: `Hello, world!`

As in other programming lectures, we, too, shall start with a "Hello world" program. Let us say in C.

helloworld.c

```c
#include <stdio.h>

int main(int argc, char *argv[])
{
   printf("Hello,␣world\n");
   return 0;
}
```

- We can immediately see that this program always finishes and the first twelve characters it outputs are *Hello, world*.
- This is not the only way how to write a program with the same functionality…

Program Hello, world! (2nd version)

helloworld2.c

```
#include <stdio.h>

int exp(int i, int n)
/* Returns the n-th power of i */
{
   int pow, j;
   pow=1;
   for (j=1; j<=n; ++j) pow *= i;
   return pow;
}
```

Program Hello, world! (2nd version)

```
int main(int argc, char *argv[]) {
    int n, total, x, y, z;
    scanf("%d", &n);
    total=3;
    while (1) {
        for (x=1; x<=total-2; ++x) {
            for (y=1; y<=total-x-1; ++y) {
                z=total-x-y;
                if (exp(x,n)+exp(y,n)==exp(z,n)) {
                    printf("Hello,␣world\n");
                    return 0;
                }
            }
        }
        ++total;
    }
}
```

In which cases `helloworld2` finishes with the first twelve characters it outputs being *Hello, world*?

Program `helloworld2` finishes and the first twelve characters it outputs are *Hello, world*, if and only if `scanf` reads a number $n \leq 2$. For $n > 2$ program `helloworld2` will not finish its computation.

Proof of this fact is equivalent to proving the Fermat's Last Theorem!

Problem HELLOWORLD

Helloworld

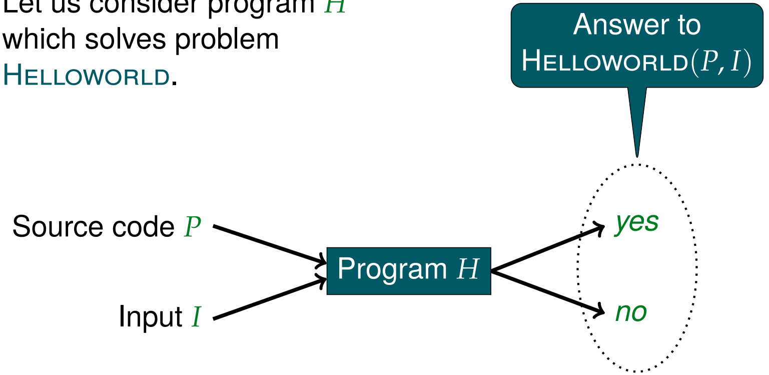
Instance:	Source code of program P in language C and input file I.
Question:	Is it true that the first 12 characters which are output by P on input I are *Hello, world*? (Finishing is not required.)

? Is it possible to write a program H in language C which given source code P and I answers the question of problem HELLOWORLD?

! We shall show it is not possible.

Undecidability of HELLOWORLD

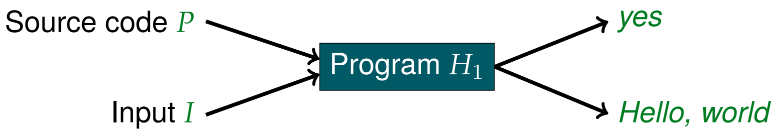
Let us consider program H which solves problem HELLOWORLD.



- We assume that the input is passed to the standard input of programs P and H and is only accessed by function `scanf`.
- We assume that the output is written to the standard output only by function `printf`.

Say hello instead of rejecting

We shall modify program H (to H_1) so that instead of *no* it outputs *Hello, world*.



The following simple modification gives us program H_1:

If the first character written by H to the standard output is *n*, we know H will write *no* eventually. We can thus modify `printf` so that *Hello, world* is output instead.

What can H_1 say about itself?

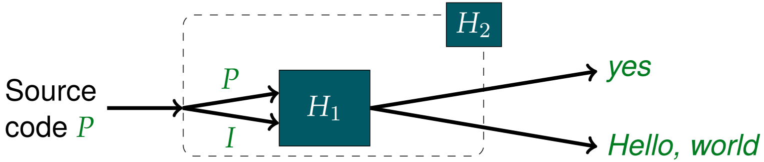
What can program H_1 say about itself?

H_1 expects a source code P and an input file I. Thus we cannot pass H_1 directly to H_1 (there is no input file to pass).

We have to modify H_1 so that it expects only one input file which is used as both source code P and input file I.

Two inputs in one

Program H_2 expects only one input file which is passed to H_1 as both the source code P and input file I.



1. Program H_2 first reads the whole input and stores it in array A which is allocated in memory (e.g. using `malloc`).
2. After that H_1 is simulated, whereas:
   a. When H_1 reads input using `scanf`, H_2 uses array A (i.e. `scanf` is replaced with reading from A).
   b. Two indices in array A are used to remember where in P and I is H_1 currently reading.

Let us ask H_2 about itself…

$$\begin{tikzcd}
 & \text{If } H_2 \text{ receives as the input} \\ \text{the source code of } H_2 \arrow[d] & \\
 & \boxed{H_2} \arrow[dl] \arrow[dr] & \\
\textit{yes} \arrow[dd, Leftrightarrow] & \text{answers} & \textit{Hello, world} \arrow[dd, Leftrightarrow] \\
 & \text{if and only if} & \\
H_2 \text{ outputs } \textit{Hello, world} \arrow[dr] & & H_2 \text{ does not output } \textit{Hello, world} \arrow[dl] \\
 & \lightning &
\end{tikzcd}$$

So what?

- $\Rightarrow$ Program H_2 cannot exist.
- $\Rightarrow$ Program H_1 cannot exist.
- $\Rightarrow$ Program H cannot exist.
- $\Rightarrow$ Problem HELLOWORLD cannot be solved by any program in C (and is thus algorithmically undecidable).

Calling function **`foo`**

Let us consider the following problem.

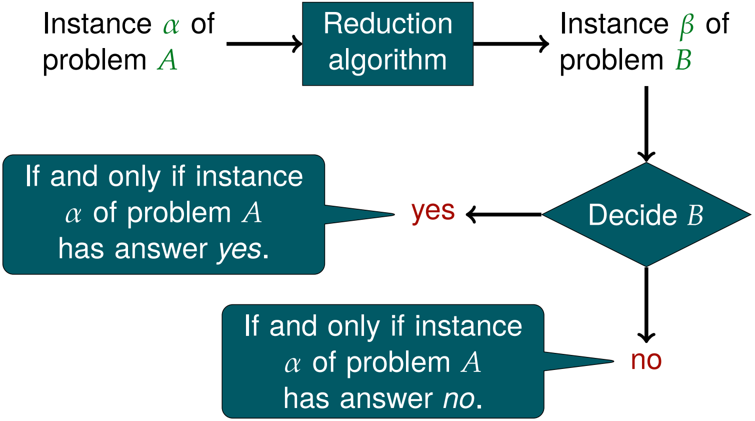
Calling function `foo`

Instance:	Source code of program Q in C and input file V.
Question:	Does program Q call function named **`foo`** when working on input I?

- We want to show that problem CALLING FUNCTION **`foo`** is algorithmically undecidable.
- We shall show that if we would be able to decide problem CALLING FUNCTION **`foo`**, we would be able to decide problem HELLOWORLD as well.

A light introduction to reducibility

If we are able to solve problem A using a solver for problem B we say that A is reducible to B.


Instance α of problem A
Reduction algorithm
Instance β of problem B
Decide B
yes
no
If and only if instance α of problem A has answer *yes*.
If and only if instance α of problem A has answer *no*.

Call "Hello, world"

- We shall reduce problem HELLOWORLD to problem CALLING FUNCTION **`foo`**.
- We shall describe how to transform an instance of problem HELLOWORLD (program P and input I) into an instance of problem CALLING FUNCTION **`foo`** (program Q and input V).
- We have to ensure that

program P with input I writes *Hello, world* as the first 12 characters of the output,

if and only if

program Q with input V calls function named `foo`.

- Problem CALLING FUNCTION **`foo`** is thus algorithmically undecidable.

How to make a call from a greeting

The input to the reduction is program P and input file I.

1. If P contains function `foo`, we rename it and its calls (refactoring, the modified program is called P_1).
2. Add function named `foo` to P_1, it does nothing and is not called ($\rightarrow P_2$).
3. Modify P_2 so that it stores the first 12 characters it outputs in array A ($\rightarrow P_3$).
4. Modify P_3 so that when it uses an output command, it first checks, whether the first 12 characters in A are *Hello, world*. If so, it calls function `foo`.
5. The last step above gives us the required program Q and input $V = I$.

Disadvantages of C for computability theory

- C language is too complicated.
- We would have to define a model of computation (i.e. generalized computer) for the C language.
- At the time of origins of computability theory, no computers or higher level languages were available and thus the theory is usually built using more traditional tools.
- We need a simple computation model, which would be powerful enough to capture our intuitive notion of an algorithm.

A bit of history …

10th Hilbert's problem

In year 1900, David Hilbert formulated 23 problems. The 10th problem can be formulated as follows.

Given a Diophantine equation with any number of unknown quantities and with rational integral numerical coefficients: To devise a process according to which it can be determined in a finite number of operations whether the equation is solvable in rational integers.

To answer this question a formal notion of an algorithm and effective computability was needed.

Intuitively: An algorithm is a finite sequence of simple instructions which leads to a solution of given problem.

Church’s thesis

In year 1934, Alonzo Church proposed the following thesis:

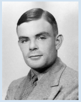
Effectively computable functions are exactly those which are λ-definable.

Later (1936) he revised the thesis in the following way.

Effectively computable functions are exactly those which are partially recursive.

Turing's thesis

In year 1936, Alan Turing proposed the following thesis



To every algorithm in intuitive sense we can construct a Turing machine which implements it.

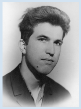
- The above mentioned models of computation (λ-calculus, partially recursive functions, Turing machines) define the same class of algorithmically computable functions.
- The above thesis is usually refered to as Church-Turing thesis.

10th Hilbert's problem

Hilbert's 10th problem can be restated as follows.

Find an algorithm to determine whether a given polynomial Diophantine equation with integer coefficients has an integer solution.

In year 1970, Yuri Matiyasevich gave a negative answer.



There is no algorithm which would determine whether a given polynomial Diophantine equation with integer coefficients has an integer solution.

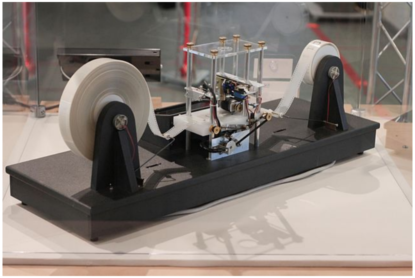
Equivalent computation models

According to Church-Turing thesis, intuitive notion of algorithm is also equivalent with…

- description of a Turing machine,
- program for RAM,
- derivation of a partial recursive function,
- derivation of a function in λ-calculus,
- program in a higher level programming language, such as C, Pascal, Java, Basic etc.,
- program in a functional programming language such as Lisp, Haskell etc.

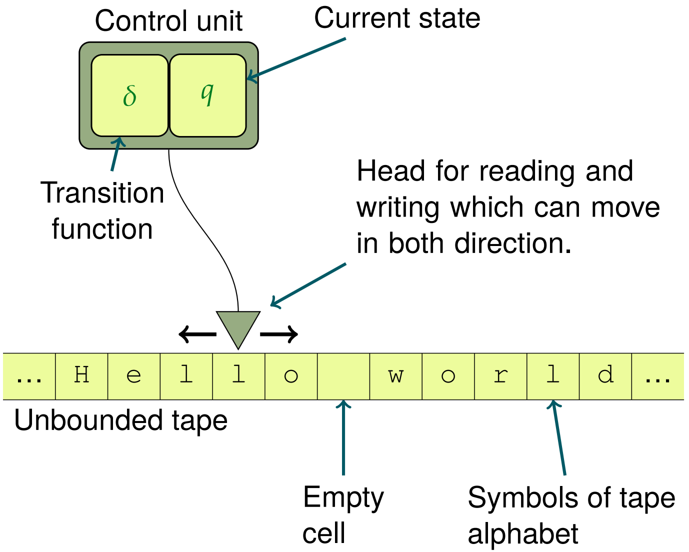
In all these models we can compute the same functions and solve the same problems.

Turing machines



By Rocky Acosta — Own work, CC BY 3.0

Turing machine



Turing machine (definition)

(1-tape deterministic) Turing machine (TM) M is a quintuple

$$M = (Q, \Sigma, \delta, q_0, F)$$

- Q is a finite set of states.
- Σ is a finite tape alphabet which contains character λ for an empty cell.
  - We shall often differentiate between a tape (inner) and an input (outer) alphabets.
- $\delta : Q \times \Sigma \mapsto Q \times \Sigma \times \{R, N, L\} \cup \{\perp\}$ is a transition function, where $\perp$ denotes an undefined transition.
- $q_0 \in Q$ is an initial state.
- $F \subseteq Q$ is a set of accepting states.

Configuration and display of a Turing machine

- Turing machine consists of
  - a control unit,
  - a tape which is potentially infinite in both directions, and
  - a head for reading and writing which can move in both directions.
- Display is a pair (q, a), where $q \in Q$ is the current state of a Turing machine and $a \in \Sigma$ is a symbol below the head.
  - Based on display TM decides what to do next.
- Configuration captures the full state of computation of a Turing machine, it consists of
  - the current state of the control unit.
  - word on the tape (from the leftmost to rightmost empty cell), and
  - position of its head within the word on the tape.

Computation of a Turing machine

- Computation of TM M starts in the initial configuration:
  - the control unit is in the initial state,
  - the tape contains the input word, and
    - *The input word does not contain an empty cell symbol.*
  - the head is on the leftmost character of the input.
- Assume the control unit of M is in state $q \in Q$ and the head of M reads symbol $a \in \Sigma$:
- If $\delta(q,a) = \perp$ computation of M terminates,
- If $\delta(q,a) = (q',a',Z)$, where $q' \in Q$, $a' \in \Sigma$ and $Z \in \{L,N,R\}$, then M
  - changes the current state to q',
  - rewrites the symbol below the head to a', and
  - moves head one cell to left (if $Z = L$), right ($Z = R$), or the head stays at the same position ($Z = N$).

Words and languages

- Word (also string) over alphabet Σ is a finite sequence of characters $w = a_1 a_2 \dots a_k$, where $a_1, a_2, \dots, a_k \in \Sigma$.
- Length of a string $w = a_1 a_2 \dots a_k$ is denoted as $|w| = k$.
- The set of all words over alphabet Σ is denoted as Σ^*.
- Empty word is denoted as ε.
- Concatenation of words w_1 and w_2 is denoted as $w_1 w_2$.
- Language $L \subseteq \Sigma^*$ is a set of words over alphabet Σ.
- Complement of language L is denoted as $\overline{L} = \Sigma^* \setminus L$.
- Concatenation of languages L_1 and L_2 is language $L_1 \cdot L_2 = \{w_1 w_2 \mid w_1 \in L_1, w_2 \in L_2\}$.
- Kleene star operation on language L produces language $L^* = \{w \mid (\exists k \in \mathbb{N})(\exists w_1, \dots, w_k \in L)[w = w_1 w_2 \dots w_k]\}$.

A decision problem is formalized as a question whether given instance belongs to the language of positive instances.

Turing decidable languages

- TM M accepts $w \in \Sigma^*$, if computation of M with input w terminates in an accepting state.
- TS M rejects w, if computation of M with input w terminates in a state which is not accepting.
- Language of words accepted by TM M is denoted as $L(M)$.
- We denote the fact that the computation of TM M on w terminates as $M(w)\downarrow$ (computation converges).
- We denote the fact that the computation of TM M on w does not terminate as $M(w)\uparrow$ (computation diverges).
- Language L is partially (Turing) decidable (also recursively enumerable), if there is a TM M such that $L = L(M)$.
- Language L is (Turing) decidable (also recursive), if there is a TM M which always stops and $L = L(M)$.

Turing computable functions

- Each Turing machine M with tape alphabet Σ computes some partial function $f_M : \Sigma^* \mapsto \Sigma^*$.
- If $M(w)\downarrow$ for a given input $w \in \Sigma^*$, the value $f_M(w)$ is defined which is denoted as $f_M(w)\downarrow$.
- The value of $f_M(w)$ is then the word on an (output) tape of $M(w)$ after the computation terminates.
- If $M(w)\uparrow$, then the value $f_M(w)$ is undefined, which is denoted as $f_M(w)\uparrow$.
- Function $f : \Sigma^* \mapsto \Sigma^*$ is Turing computable, if there is a Turing machine M which computes it.

To each Turing computable function there is infinitely many Turing machines computing it!

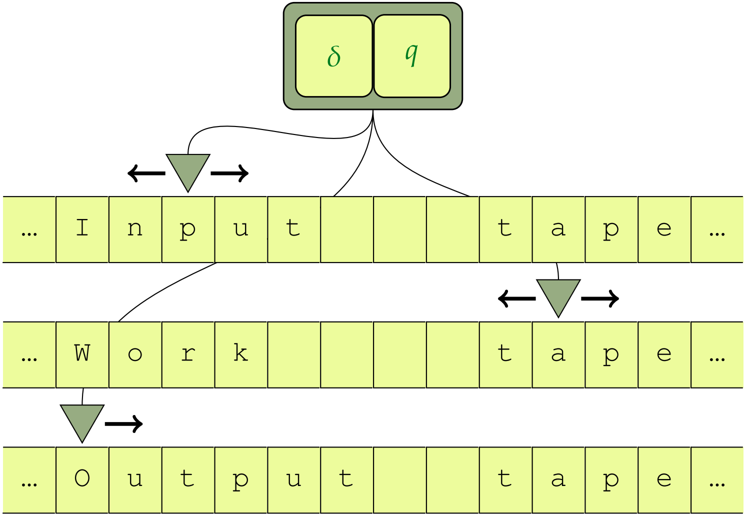
Variants of Turing machines

Turing machines have a lot of variants, for example

- TM with a tape potentially infinite only in one direction.
- TM with multiple tapes (we can differentiate input/output/work tapes).
- TM with multiple heads on tapes,
- TM with only binary alphabet,
- Nondeterministic TM's.

All these variants are equivalent to "our" model.

Structure of a 3-Tape Turing Machine



Multitape Turing Machine

k-Tape Turing Machine differs from a single tape Turing machine as follows:

- It has k tapes with a head on each of them.

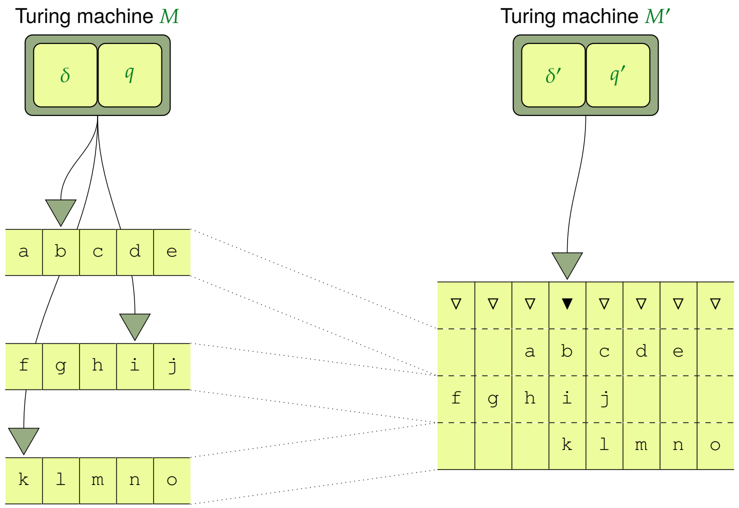
Input tape	contains the input at the beginning. *Often read-only.*
Work tapes	are read-write.
Output tape	at the end contains the output string. *Often write-only with head moving only to the right.*

- Heads on tapes move independently on each other.
- Transition function is defined as $\delta : Q \times \Sigma^k \mapsto Q \times \Sigma^k \times \{R, N, L\}^k \cup \{\perp\}$.

Theorem 1

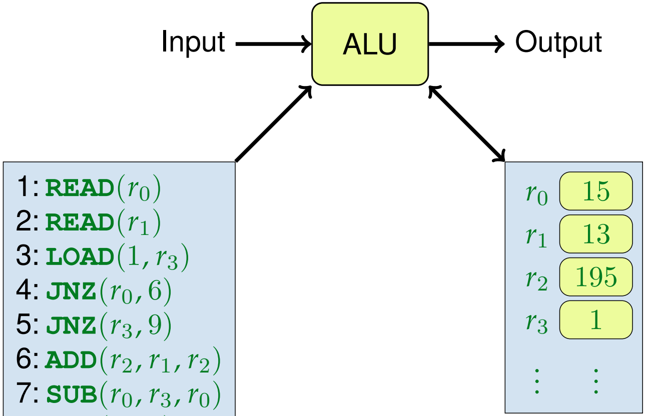
To each k-tape Turing machine M there is a single tape Turing machine M', which simulates the computation of M, accepts the same language and computes the same function as M.

Representation of k tapes on a single tape



Random Access Machine

Random Access Machine (RAM)



```
1: READ$(r_0)$
2: READ$(r_1)$
3: LOAD$(1, r_3)$
4: JNZ$(r_0, 6)$
5: JNZ$(r_3, 9)$
6: ADD$(r_2, r_1, r_2)$
7: SUB$(r_0, r_3, r_0)$
8: JNZ$(r_0, 6)$
9: PRINT$(r_2)$
```

Program

Memory
split into an unbounded number of registers

Random Access Machine (definition)

- Random Access Machine (RAM) consists of
    - a control unit (processor, CPU), and
    - an unbounded memory.
- Memory of RAM is split into registers which we shall denote as r_i, $i \in \mathbb{N}$.
- A register can store any natural number (0 initially).
- Number stored in register r_i shall be denoted as $[r_i]$.
- Indirect addressing: $[[r_i]] = [r_{[r_i]}]$.
- Program for RAM is a finite sequence of instructions $P = I_0, I_1, \ldots, I_\ell$.
- Instructions are executed in the order given by the program.

Possible RAM instructions

Instruction	Effect
LOAD(C, r_i)	$r_i \leftarrow C$
ADD(r_i, r_j, r_k)	$r_k \leftarrow [r_i] + [r_j]$
SUB(r_i, r_j, r_k)	$r_k \leftarrow [r_i] \dot{-} [r_j]$
COPY$([r_p], r_d)$	$r_d \leftarrow [[r_p]]$
COPY$(r_s, [r_d])$	$r_{[r_d]} \leftarrow [r_s]$
JNZ(r_i, I_z)	**if** $[r_i] > 0$ **then goto** z
READ(r_i)	$r_i \leftarrow$ input
PRINT(r_i)	output $\leftarrow [r_i]$

$$x \dot{-} y = \begin{cases} x - y & x > y \\ 0 & \text{otherwise} \end{cases}$$

Languages decidable with RAM

- Consider alphabet $\Sigma = \{\sigma_1, \sigma_2, \ldots, \sigma_k\}$.
- We pass a string $w = \sigma_{i_1}\sigma_{i_2}\ldots\sigma_{i_n}$ to RAM R as a sequence of numbers $i_1, \ldots, i_n$.
- End of the input can R recognize because **READ** returns 0 when no more input is available.
- RAM R accepts w, if $R(w)\downarrow$ and the first number written to the output by R is1.
- RAM R rejects w, if $R(w)\downarrow$ and R either does not output any number or the first number written to the output is not 1.
- Language of strings accepted by RAM R is denoted as $L(R)$.
- If language $L = L(R)$ for some RAM R, then it is partially decidable (with RAM).
- If moreover this R terminates with every input, then we say that $L = L(R)$ is decidable (with RAM).

Arithmetic functions computable with RAM

We say that RAM R computes a partial arithmetic function $f : \mathbb{N}^n \mapsto \mathbb{N}$, $n \geq 0$, if with input n-tuple $(x_1, \ldots, x_n)$:

- If $f(x_1, \ldots, x_n)\downarrow$, then $R(x_1, \ldots, x_n)\downarrow$ and R outputs value $f(x_1, \ldots, x_n)$.
- If $f(x_1, \ldots, x_n)\uparrow$, then $R(x_1, \ldots, x_n)\uparrow$.

Function f which is computable by some RAM R is called RAM computable.

String functions computable with RAM

RAM R computes a partial function $f : \Sigma^* \mapsto \Sigma^*$, where $\Sigma = \{\sigma_1, \sigma_2, \dots, \sigma_k\}$, if the following is satisfied:

- The input strring $w = \sigma_{i_1}\sigma_{i_2} \dots \sigma_{i_n}$ is passed as a sequence of numbers $i_1, \dots, i_n$.
- End of the input can R recognize because **READ** returns 0 when no more input is available.
- If $f(w){\downarrow} = \sigma_{j_1}\sigma_{j_2} \dots \sigma_{j_m}$, then $R(w){\downarrow}$ and it writes numbers $j_1, j_2, \dots, j_m, 0$ to the output.
- If $f(w){\uparrow}$, then $R(w){\uparrow}$.

Function f which is computable by some RAM R is called RAM computable.

Programming on RAM

Programs for RAM corresponds to a procedural language:

- We can use variables (scalar and unbounded arrays).
- Cycles (for and while) — using conditional jump and a counter variable.
- Uncoditional jump (goto) — using an auxiliary register where we store 1 and use a conditional jump.
- Conditional statement — using a conditional jump.
- Functions and procedures — we can inline a body of a function directly to the place where it is used (*inline*)
- We don't have recursive calls — we can implement them using a cycle while and a stack.

Variables in a program for RAM

Assume that we use arrays $A_1, \ldots, A_p$ and scalar variables $x_0, \ldots, x_s$.

- Arrays are indexed with natural numbers (that is starting from 0).
- Element $A_i[j]$, where $i \in \{1, \ldots, p\}, j \in \mathbb{N}$, is stored in register $r_{i+j*(p+1)}$.
- Elements of array A_i, $i = 1, \ldots, p$ are thus stored in registers $r_i, r_{i+p+1}, r_{i+2(p+1)}, \ldots$.
- A scalar variable x_i, where $i \in \{0, \ldots, s\}$ is stored in register $r_{i*(p+1)}$.
- Scalar variables are thus stored in registers $r_0, r_{p+1}, r_{2(p+1)}, \ldots$.

Turing machine $\longrightarrow$ RAM

Theorem 2

To each Turing machine M there is an equivalent RAM R.

- Content of the tape is stored in two arrays:
  - T_r contains the right hand side and
  - T_l contains the left hand side.
- Position of head — index in variable h and side (right/left) in variable s
- State — variable q.
- Choosing instruction — conditional statement based on h, s, and q.

RAM $\longrightarrow$ Turing machine

Theorem 3

To each RAM R there is an equivalent Turing machine M.

Content of the memory of R is represented on a tape of M as follows:

If the currently used registers are $r_{i_1}, r_{i_2}, \ldots, r_{i_m}$, where $i_1 < i_2 < \cdots < i_m$, then the tape contains string:

$$(i_1)_B|([r_{i_1}])_B\#(i_2)_B|([r_{i_2}])_B\#\ldots\#(i_m)_B|([r_{i_m}])_B$$

RAM $\longrightarrow$ Turing machine (structure of TM)

We shall describe a 4-tape TM M to a RAM R.

Input tape: sequence of numbers passed to R as the input. Numbers are written in binary and separated with #. M only reads this tape.

Output tape: M writes here the numbers output by R. They are written in binary and separated with #. M only writes to this tape.

Memory of RAM: content of memory of R.

Auxiliary tape: for computing addition, subtraction, indirect indices, copying part of the memory tape, etc.

Numbering Turing machines

How to number Turing machines

Our goal is to assign a natural number to each Turing machine.

1. Encode a Turing machine as a string over a small alphabet.
2. Encode any string over Γ as a binary string.
3. Assign a number to each binary string.
4. The number we get in this way for a given Turing machine is called a Gödel number.

A few technical restrictions

We shall restrict to Turing machines which

1. have a single accepting state and
2. use only binary input alphabet $\Sigma_{in} = \{0, 1\}$.

- Restriction on the input alphabet means that the input strings are passed to Turing machines only as sequences of 0-es and 1-s.
- Work alphabet is not restricted — during its computation a Turing machine can write any symbols to a tape.
- Any finite alphabet can be encoded in binary.
- Any Turing machine M can be easily modified into a Turing machine satisfying these restrictions.

Encoding transition function

$$\begin{tikzcd}
M = (Q, \Sigma, \delta, q_0, F = \{q_1\}) \arrow[d] \\
\boxed{\begin{array}{c}\delta \text{ is written as a string } v_M \text{ in alphabet} \\ \Gamma = \{0, 1, L, N, R, |, \#, ;\}\end{array}} \arrow[d, "v_M"] \\
\boxed{\text{Each characters of } v_M \text{ is encoded with } 3 \text{ bits}} \arrow[d] \\
\langle M \rangle
\end{tikzcd}$$

Binary string representing M

Encoding transition function in alphabet Γ

- Assume that
  - $Q = \{q_0, q_1, \ldots, q_r\}$ for some $r \geq 1$, where q_0 is the initial state and q_1 is the accepting state.
  - $\Sigma = \{X_0, X_1, X_2, \ldots, X_s\}$ for some $s \geq 2$, where $X_0 = `0'$, $X_1 = `1'$, and $X_2 = `\lambda'$.
- Instruction $\delta(q_i, X_j) = (q_k, X_l, Z)$, where $Z \in \{L, N, R\}$ is encoded as string

$$(i)_B|(j)_B|(k)_B|(l)_B|Z \quad .$$

- If $C_1, \ldots, C_n$ are the codes of all instructions of TM M, then the transition function δ is encoded as

$$C_1\#C_2\#\ldots\#C_n \quad .$$

Write it in binary

$$\begin{tikzcd}
\lambda \arrow[d] \\
\delta(q_3, X_7) = (q_5, X_2, R) \arrow[d] \\
11|111|101|10|R
\end{tikzcd}$$

Γ	kód
0	000
1	001
L	010
N	011
R	100
|	101
#	110
;	111

Characters of alphabet Γ are encoded using this table.

001001101001001001101001000001101001000101100

Numbering binary strings

- Given a binary string $w \in \{0,1\}^*$ we assign it number i such that $(i)_B = 1w$.
- String with number i is denoted as w_i (i.e. $(i)_B = 1w_i$).
- We get a 1–1 correspondence (bijection) between $\{0,1\}^*$ and positive natural numbers.
- Moreover we shall assume that 0 corresponds to an empty string, i.e. $w_0 = w_1 = \varepsilon$.

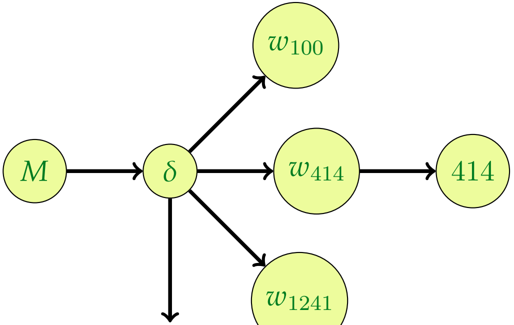
w_i	$1w_i$	i
ε	1	1
0	10	2
1	11	3
00	100	4
$\vdots$	$\vdots$	$\vdots$
001011	1001011	75
$\vdots$	$\vdots$	$\vdots$

Gödel number

- We can associate a Gödel number e with a Turing machine M such that w_e is an encoding of M.
- Turing machine with Gödel number e is denoted as M_e.
- If string w_e is not a syntactically correct encoding of a Turing machine, then M_e is an empty Turing machine which immediately rejects every input and thus $L(M_e) = \emptyset$.
- Now we can assign a Turing machine M_e to every natural number e.

Disambiguity of encodings of TMs

- Encoding of a TM is not unique because it depends on
  - the order of instructions,
  - numbering of states (except initial and accepting),
  - numbering of characters of tape alphabet (except 0, 1, λ),
  - binary encoding of a number can contain any number of leading 0s.
- Every TM has actually an infinite number of possible encodings and infinitely many Gödel numbers.



One of the Gödel numbers of M

Encoding of objects (notation)

- Every object (e.g. number, string, Turing machine, RAM, graph, or formula) can be encoded into a binary string.
- We can encode n-tuples of objects as well.

Definition 4

- $\langle X \rangle$ denotes a binary string encoding object X.
- $\langle X_1, \dots, X_n \rangle$ denotes a binary string encoding n-tuple of objects $X_1, \dots, X_n$.

- For example if M is a Turing machine, then $\langle M \rangle$ denotes a binary string which encodes M.
- If M is a Turing machine and x is a string, then $\langle M, x \rangle$ denotes a binary string encoding the pair of M and x.

Universal Turing machine

Universal Turing machine

- The input to a universal Turing machine $\mathcal{U}$ is a pair $\langle M, x\rangle$, where M is a Turing machine and x is a string.
- $\mathcal{U}$ simulates computation of M on input x.
- The result of $\mathcal{U}(\langle M, x\rangle)$ (i.e. terminating/accepting/rejecting and contents of the output tape) is given by the result of computation $M(x)$.
- For simplicity, we shall describe $\mathcal{U}$ as a 3-tape TM.
- We can transform it into a single tape universal Turing machine.
- The language of $\mathcal{U}$ is called universal language and it is denoted as L_u that is

$$L_u = L(\mathcal{U}) = \{\langle M, x\rangle \mid x \in L(M)\}.$$

Structure of a universal Turing machine

1st tape contains the input of $\mathcal{U}$ that is the code $\langle M, x\rangle$.

$$\langle M, x\rangle$$

2nd tape contains the work tape of M. Symbols X_i are encoded as $(i)_B$ in blocks of the same length separated with `|`.

```
... |010|001|100|000|010|011| ...
```

3rd tape contains the number of the current state q_i of M.

$$\texttt{10011}\,(=(i)_B)$$

Algorithmically (un)decidable languages

Definition

Definition 5

- Language L is partially decidable, if it is accepted by some Turing machine M (i.e. $L = L(M)$).
- Language L is decidable, if there is a Turing machine M which accepts L (i.e. $L = L(M)$) and moreover its compution over any input x stops (i.e. $M(x)\downarrow$).

- Partially decidable languages = recursively enumerable languages.
- Decidable languages = recursive languages.

Basic properties of decidable languages

Theorem 6

If L_1 and L_2 are (partially) decidable languages, then $L_1 \cup L_2$, $L_1 \cap L_2$, $L_1 \cdot L_2$, L_1^* are also (partially) decidable languages.

Theorem 7 (Post's theorem)

Language L is decidable if and only if L and $\overline{L}$ are both partially decidable languages.

1. Are all languages over a finite alphabet at least partially decidable?
2. Are all partially decidable languages also decidable?

How many decidable languages are there?

Definition 8

Set A is countable if there is a 1–1 function $f : A \mapsto \mathbb{N}$, i.e. if the elements of A can be numbered.

- There is only countable many Turing machines — each has a Gödel number.
- Each partially decidable language is accepted by some Turing machine.

Lemma 9

There is only countable many partially decidable languages.

Are all languages decidable?

A language $L \subseteq \{0,1\}^*$ corresponds to a set of natural numbers

$$A = \{i - 1 \mid i \in \mathbb{N} \setminus \{0\} \wedge w_i \in L\}.$$

- $\mathcal{P}(\mathbb{N})$ is uncountable.
- There is uncountable many languages over alphabet $\{0,1\}$.

There must be languages over alphabet $\{0,1\}$ which are not partially decidable!

We could even say that most of the languages are not partial decidable.

Diagonal language

Let us define the diagonal language as follows.

$$\mathrm{DIAG} = \{\langle M\rangle \mid \langle M\rangle \notin L(M)\}$$

Its complement is then defined as

$$\overline{\mathrm{DIAG}} = \{\langle M\rangle \mid \langle M\rangle \in L(M)\}$$

Theorem 10

1. *Language* DIAG *is not partially decidable (it is not recursively enumerable).*
2. *Language* $\overline{\mathrm{DIAG}}$ *is not decidable, but it is partially decidable.*

Universal language

- Problem of deciding whether a given string y belongs to the universal language L_u is a formalization of the UNIVERSAL PROBLEM:

Universal problem

Instance:	Code of Turing machine M and an input x.
Question:	Is $x \in L(M)$? In other words, does M accept input x?

Theorem 11

Universal language (and UNIVERSAL PROBLEM) is partially decidable but it is not decidable.

Halting problem

- A classical example of an algorithmically undecidable problem is the HALTING PROBLEM.

Halting problem	
Instance:	Code of Turing machine M and input x.
Question:	Is $M(x)\downarrow$? In other words does the computation of M over input x terminate?

Theorem 12

HALTING PROBLEM is partially decidable but it is not decidable.

Algorithmically computable function

Functions — notation

Let $f, g : \Sigma^* \mapsto \Sigma^*$ be two partial functions, then

- Domain of f is the set

$$\operatorname{dom} f = \{x \in \Sigma^* \mid f(x){\downarrow}\}$$

- Range of f is the set

$$\operatorname{rng} f = \{y \in \Sigma^* \mid (\exists x \in \Sigma^*)[f(x){\downarrow}= y]\}$$

- f and g are conditionally equal ($f \simeq g$) if

$$f \simeq g \iff \left[\operatorname{dom} f = \operatorname{dom} g \text{ and } (\forall x \in \operatorname{dom} f)[f(x) = g(x)]\right]$$

Algorithmically computable function

Intuitively: (Algorithmically) computable function is a function computable by some algorithm.

Definition 13

- A partial function $f : \Sigma^* \mapsto \Sigma^*$ is (algorithmically) computable if it is Turing computable.
- φ_e denotes the function computed by Turing machine M_e.

- Computable functions = partial recursive functions.
- Total computable functions = (total) recursive functions.
- We also consider functions of multiple parameters and arithmetic functions, e.g. $f(x, y) = x^2 + y^2$ corresponds to a function over strings $f'(\langle x, y\rangle) = \langle x^2 + y^2\rangle$.
- There is only countable many computable functions $\Longrightarrow$ not all functions are computable.

Universal function

Theorem 14

A universal function Ψ for computable functions satisfying

$$\Psi(\langle e, x\rangle) \simeq \varphi_e(\langle x\rangle)$$

is a computable function.

…because we have a universal Turing machine.

Properties of (partially) decidable languages

Partially decidable languages

Theorem 15

Given a language $L \subseteq \Sigma^*$, the following are equivalent:

1. L *is partially decidable.*
2. *There is a Turing machine* M *satisfying*

$$L = \{ x \in \Sigma^* \mid M(x)\downarrow \}.$$

3. *There is an algorithmically computable function* $f(x)$ *satisfying*

$$L = \operatorname{dom} f = \left\{ x \in \Sigma^* \mid f(x)\downarrow \right\}$$

4. *There is a decidable language* B *satisfying*

$$L = \left\{ x \in \Sigma^* \mid (\exists y \in \Sigma^*)[\langle x, y \rangle \in B] \right\}$$

Decidable languages

Theorem 16

Language $L \subseteq \Sigma^*$ is decidable if and only if its characteristic function

$$\chi_L(x) = \begin{cases} 1 & x \in L \\ 0 & x \notin L \end{cases}$$

is computable.

Strings ordering

Definition 17 (Lexicographic order)

Let Σ be an alphabet and let us assume that $<$ is a strict order over characters in Σ. Let $u, v \in \Sigma^*$ be two different strings. We say that u is lexicographically smaller than v if

1. u is shorter (i.e. $|u| < |v|$), or
2. both strings have the same length (i.e. $|u| = |v|$) and if i is the first index with $u[i] \neq v[i]$, then $u[i] < v[i]$.

This fact is denoted with $u \prec v$. As usual we extend this notation to $u \preceq v$, $u \succ v$ and $u \succeq v$.

Enumerating

An enumerator for a language L is a Turing machine E, which

- ignores its input,
- during its computation writes strings $w \in L$ to a special output tape separated with '$\#$', and
- each string $w \in L$ is eventually written by E.
- If L is infinite, E never stops.

Theorem 18

1. *Language L is partially decidable if and only if there is an enumerator E for L.*
2. *Language L is decidable if and only if there is an enumerator E for L which outputs elements of L in lexicographic order.*

Enumerating and functions

Theorem 19

Let L be an infinite language, then L is …

1. *…partially decidable, if and only if it is a range of a total algorithmically computable function f (i.e. $L = \operatorname{rng} f$).*
2. *…decidable, if and only if it is a range of an increasing total algorithmically computable function f (i.e. $L = \operatorname{rng} f$).*

- Function $f : \Sigma^* \longrightarrow \Sigma^*$ is increasing, if $u \prec v$ implies $f(u) \prec f(v)$ for every pair of strings $u, v \in \Sigma^*$ where $f(u)\downarrow$ and $f(v)\downarrow$.

Reducibility and completeness

Reducibility and completeness

Definition 20

A language A is m-reducible to a language B (which is denoted as $A \leq_m B$) if there is a total computable function f s.t.

$$(\forall x \in \Sigma^*)[x \in A \iff f(x) \in B]$$

Language A is m-complete if A is partially decidable and any partially decidable language B is m-reducible to A.

- 1-reducibility and 1-completeness — we require the function f to be moreover 1–1.
- $\leq_m$ is a reflexive and transitive relation (it is a quasiorder).
- If $A \leq_m B$ and B is (partially) decidable, then so is A.
- If $A \leq_m B$, B is partially decidable and A is m-complete, then B is also m-complete.

Complete languages

The Halting problem can be formalized as language

$$\mathrm{HALT} = \{\langle M, x\rangle \mid M(x)\downarrow\}$$

Theorem 21

L_u, DIAG, and HALT are m-complete languages. In particular, they are partially decidable, but not decidable.

Rice's theorem

Theorem 22 (Rice's theorem (languages))

Let C be a class of partially decidable languages and let us define $L_C = \{\langle M\rangle \mid L(M) \in C\}$. Then language L_C is decidable if and only if C is either empty or it contains all partially decidable languages.

Theorem 23 (Rice's theorem (functions))

Let C be a class of computable functions and let us define $A_C = \{w_e \mid \varphi_e \in C\}$. Then language A_C is decidable if and only if C is either empty or it contains all computable functions.

Rice's theorem (consequences)

Rice's theorem implies that the following languages are undecidable:

$$\begin{aligned}
K_1 &= \{\langle M\rangle \mid L(M) \neq \emptyset\} \\
\mathrm{Fin} &= \{\langle M\rangle \mid L(M) \text{ is a finite language }\} \\
\mathrm{Cof} &= \{\langle M\rangle \mid \overline{L(M)} \text{ is a finite language}\} \\
\mathrm{Inf} &= \{\langle M\rangle \mid L(M) \text{ is an infinite language}\} \\
\mathrm{Dec} &= \{\langle M\rangle \mid L(M) \text{ is a decidable language}\} \\
\mathrm{Tot} &= \{\langle M\rangle \mid L(M) = \Sigma^*\} \\
\mathrm{Reg} &= \{\langle M\rangle \mid L(M) \text{ is a regular language}\}
\end{aligned}$$

Post correspondence problem

Post correspondence problem (PCP)

Instance: Collection P of “dominos” (pairs of strings):

$$P = \left\{ \left[\frac{t_1}{b_1} \right], \left[\frac{t_2}{b_2} \right], \ldots, \left[\frac{t_k}{b_k} \right] \right\}$$

where $t_1, \ldots, t_k, b_1, \ldots, b_k \in \Sigma^*$.

Question: Is there a matching sequence $i_1, i_2, \ldots, i_l$ where $l \geq 1$ and $t_{i_1} t_{i_2} \ldots t_{i_l} = b_{i_1} b_{i_2} \ldots b_{i_l}$?

Theorem 24

POST CORRESPONDENCE PROBLEM is undecidable.

S-m-n theorem

Theorem 25 (s-m-n)

For any two natural numbers $m, n \geq 1$ there is a 1–1 total computable function $s_n^m : \mathbb{N}^{m+1} \to \mathbb{N}$ such that for every $x, y_1, y_2, \ldots, y_m, z_1, \ldots, z_n \in \Sigma_b^*$:

$$\varphi^{(n)}_{s_n^m(x, y_1, y_2, \ldots, y_m)}(z_1, \ldots, z_n) \simeq \varphi_x^{(m+n)}(y_1, \ldots, y_m, z_1, \ldots, z_n)$$

Complexity

Basic complexity classes

Decision problems

- In a decision problem we want to decide whether a given instance x satisfies a specified condition.
- Answer is *yes*/*no*.
- Formalized as a language $L \in \Sigma^*$ of positive instances and a decision whether $x \in L$.
- Examples of decision problems:
  - Is a given graph connected?
  - Does a given logical formula have a model?
  - Does a given linear program admit a feasible solution?
  - Is a given number prime?

Search and optimization problems

- In a search problem we aim to find for a given instance x an output y which satisfies a specified condition.
- Answer is y or information that no suitable y exists.
- Formalized as a relation $R \subseteq \Sigma^* \times \Sigma^*$.
- Examples of search problems:
  - Find all strong components of a directed graph.
  - Find a satisfying assignment to a logical formula.
  - Find a feasible solution to a given linear program.
- In an optimization problem we moreover require the output y to be maximal or minimal with respect to some measure.
- Examples of optimization problems:
  - Find a maximum flow in a network.
  - Find a shortest path in a graph.
  - Find an optimum solution to a linear program.

Time and space complexity of a Turing machine

Definition 26

Let M be (deterministic) Turing machine and let $f : \mathbb{N} \mapsto \mathbb{N}$ be a function.

- We say that M runs (or works) in time $f(n)$ if for any string x of length $|x| = n$ the computation of M over x terminates within $f(n)$ steps.
- We say that M works in space $f(n)$ if for any string x of length $|x| = n$ the computation of M over x terminates and uses at most $f(n)$ tape cells.

Basic deterministic complexity classes

Definition 27

Let $f : \mathbb{N} \mapsto \mathbb{N}$ be a function, then we define classes:

$\mathrm{TIME}(f(n))$ class of languages which can be accepted by Turing machines running in time $O(f(n))$.

$\mathrm{SPACE}(f(n))$ class of languages which can be accepted by Turing machines working in space $O(f(n))$.

- Trivially, $\mathrm{TIME}(f(n)) \subseteq \mathrm{SPACE}(f(n))$ for any function $f : \mathbb{N} \mapsto \mathbb{N}$.

Notable deterministic complexity classes

Definition 28

Class of problems solvable in polynomial time:

$$\mathrm{P} = \bigcup_{k\in\mathbb{N}} \mathrm{TIME}(n^k)$$

Class of problems solvable in polynomial space:

$$\mathrm{PSPACE} = \bigcup_{k\in\mathbb{N}} \mathrm{SPACE}(n^k).$$

Class of problems solvable in exponential time:

$$\mathrm{EXPTIME} = \bigcup_{k\in\mathbb{N}} \mathrm{TIME}(2^{n^k}).$$

Why polynomials?

Thesis 29 (Strong Church-Turing thesis)

Realistic computation models can by simulated on TM with polynomial delay/space increase.

- Polynomials are closed under composition.
- Polynomials (usually) do not increase too rapidly
- Definition of P does not depend on particular computational model we use.

Thesis 30 (Cobham-Edmonds thesis, 1965)

P *roughly corresponds to the class of problems that are realistically solvable on a computer.*

Verifier

Definition 31

A verifier for a language A is an algorithm V, where

$$A = \left\{x \mid (\exists y)[V \text{ accepts } \langle x, y\rangle]\right\} .$$

- String y is also called a certificate of x.
- The time of a verifier is measured only in terms of $|x|$.
- A polynomial time verifier runs in time polynomial in $|x|$.
- It follows that if a polynomial time verifier V accepts $\langle x, y\rangle$, then y has length polynomial in the length of x.
- String y is then a polynomial certificate of x.

Class NP

Definition 32

NP is the class of languages that have polynomial time verifiers.

- Corresponds to a class of search problems where we do not how to find a solution in polynomial time but we can check if a given string is a solution to our problem.
- Languages in NP are exactly those which are accepted by nondeterministic polynomial time Turing machines.
- Nondeterminism corresponds to “guessing” the right certificate y of x.

Nondeterministic Turing machine

Nondeterministic Turing machine (NTM) is a quintuple $M = (Q, \Sigma, \delta, q_0, F)$, where

- Q, Σ, q_0, F have the same meaning as in the case of a “regular” deterministic Turing machine (DTM).
- NTM differs from a DTM in transition function, now

$$\delta : Q \times \Sigma \mapsto \mathcal{P}(Q \times \Sigma \times \{L, N, R\}).$$

- Possible views
  - M “guesses” or “chooses” the “right” transition at each step.
  - M performs all transitions at once (in parallel) and can be in many possible configurations at each time.
- Nondeterministic Turing machine is not a realistic computation model in the sense of strong Church-Turing thesis.

Language accepted by a nondeterministic TM

- Computation of NTM M over string x is a sequence of configurations $C_0, C_1, C_2, \ldots$, where
  - C_0 is an initial configuration and
  - C_{i+1} originates from C_i by applying transition function δ.
- Computation is accepting if it is finite and M is in an accepting state in the last configuration.
- String x is accepted by NTM M if there is an accepting computation of M over x.
- Language of string accepted by NTM M is denoted as $L(M)$.

Time and space complexity of NTM

Definition 33

Let M be a nondeterministic Turing machine and let $f : \mathbb{N} \mapsto \mathbb{N}$ be a function.

- We say that M works in time $f(n)$ if every computation of M over any input x of length $|x| = n$ terminates within $f(n)$ steps.
- We say that M works in space $f(n)$ if every computation of M over any input x of length $|x| = n$ terminates and uses at most $f(n)$ cells of work tape.

Basic nondeterministic complexity classes

Definition 34

Let $f : \mathbb{N} \mapsto \mathbb{N}$ be a function, then we define classes:

$\mathrm{NTIME}(f(n))$ class of languages accepted by nondeterministic TMs working in time $O(f(n))$.

$\mathrm{NSPACE}(f(n))$ class of languages accepted by nondeterministic TMs working in space $O(f(n))$.

Theorem 35

For any function $f : \mathbb{N} \mapsto \mathbb{N}$ we have that

$$\mathrm{TIME}(f(n)) \subseteq \mathrm{NTIME}(f(n)) \subseteq \mathrm{SPACE}(f(n)) \subseteq \mathrm{NSPACE}(f(n))$$

NP=nondeterministically polynomial

Theorem 36 (Alternative definition of class NP)

Class NP consists of languages accepted by nondeterministic Turing machines working in polynomial time, that is

$$\mathrm{NP} = \bigcup_{k \in \mathbb{N}} \mathrm{NTIME}(n^k).$$

TMs using sublinear space

In case of sublinear space complexity, we use multiple tapes:

Read-only input tape

Read-write working tapes

Write-only output tape (head moves only to the right)

- Only work tapes are inluded into space complexity.
- Configuration consists of
  - state,
  - position of head on the input tape,
  - position of heads on work tapes,
  - contents of work tapes.
- Configuration does not contain the input string.

Further space complexity classes

Definition 37

$$\begin{aligned} \mathrm{L} &= \mathrm{SPACE}(\log_2 n) \\ \mathrm{NL} &= \mathrm{NSPACE}(\log_2 n) \\ \mathrm{NPSPACE} &= \bigcup_{k \in \mathbb{N}} \mathrm{NSPACE}(n^k) \end{aligned}$$

Relation of space and time

Theorem 38

Let $f(n)$ be a function satisfying $f(n) \geq \log_2 n$. For any language L we have that

$$L \in \mathrm{NSPACE}(f(n)) \Rightarrow (\exists c_L \in \mathbb{N}) \left[L \in \mathrm{TIME}(2^{c_L f(n)}) \right] .$$

Corollary 39

Let $f(n)$ be a function satisfying $f(n) \geq \log_2 n$ and let $g(n)$ be a function satisfying $f(n) = o(g(n))$, then

$$\mathrm{NSPACE}(f(n)) \subseteq \mathrm{TIME}(2^{g(n)}).$$

Relations between classes

Theorem 40

The following chain of inclusions holds:

$$\mathrm{L} \subseteq \mathrm{NL} \subseteq \mathrm{P} \subseteq \mathrm{NP} \subseteq \mathrm{PSPACE} \subseteq \mathrm{NPSPACE} \subseteq \mathrm{EXPTIME}.$$

Savitch’s theorem

Savitch's theorem

Theorem 41 (Savitch's theorem)

For any function $f(n) \geq \log_2 n$ *we have that*

$$\mathrm{NSPACE}(f(n)) \subseteq \mathrm{SPACE}(f^2(n))$$

Corollary 42

$$\mathrm{PSPACE} = \mathrm{NPSPACE}$$

Hierarchy theorems

Space hierarchy theorem

Definition 43

A function $f : \mathbb{N} \to \mathbb{N}$, where $f(n) \geq \log n$, is called space constructible if the function that maps 1^n to the binary representation of $f(n)$ is computable in space $O(f(n))$.

Theorem 44 (Deterministic Space Hierarchy Theorem)

For any space constructible function $f : \mathbb{N} \to \mathbb{N}$, there exists a language A that is decidable in space $O(f(n))$ but not in space $o(f(n))$.

Space hierarchy

Corollary 45

1. *For any two functions* $f_1, f_2 : \mathbb{N} \to \mathbb{N}$*, where* $f_1(n) \in o(f_2(n))$ *and* f_2 *is space constructible,*

$$\mathrm{SPACE}(f_1(n)) \subsetneq \mathrm{SPACE}(f_2(n)).$$

2. *For any two real numbers* $0 \leq \epsilon_1 < \epsilon_2$*,*

$$\mathrm{SPACE}(n^{\epsilon_1}) \subsetneq \mathrm{SPACE}(n^{\epsilon_2}).$$

3. $\mathrm{NL} \subsetneq \mathrm{PSPACE} \subsetneq \mathrm{EXPSPACE} = \bigcup_{k\in\mathbb{N}} \mathrm{SPACE}(2^{n^k})$.

Time hierarchy theorem

Definition 46

A function $f : \mathbb{N} \to \mathbb{N}$, where $f(n) \in \Omega(n \log n)$, is called time constructible if the function that maps 1^n to the binary representation of $f(n)$ is computable in time $O(f(n))$.

Theorem 47 (Deterministic Time Hierarchy Theorem)

For any time constructible function $f : \mathbb{N} \to \mathbb{N}$, there exists a language A that is decidable in time $O(f(n))$ but not in time $o(f(n)/\log f(n))$.

Time hierarchy

Corollary 48

1. *For any two functions* $f_1, f_2 : \mathbb{N} \to \mathbb{N}$*, where* $f_1(n) \in o(f_2(n)/\log f_2(n))$ *and* f_2 *is time constructible,*

$$\mathrm{TIME}(f_1(n)) \subsetneq \mathrm{TIME}(f_2(n)).$$

2. *For any two real numbers* $1 \leq \epsilon_1 < \epsilon_2$*,*

$$\mathrm{TIME}(n^{\epsilon_1}) \subsetneq \mathrm{TIME}(n^{\epsilon_2}).$$

3. $P \subsetneq EXPTIME$.

Polynomial reducibility and $\mathbb{NP}$-completeness

Polynomial reducibility

Definition 49

Language A is polynomial time reducible to language B, written as $A \leq_m^P B$, if a polynomial time computable function $f : \Sigma^* \mapsto \Sigma^*$ exists, where

$$(\forall w \in \Sigma^*)\left[w \in A \iff f(w) \in B\right].$$

- $\leq_m^P$ is reflexive and transitive relation (it is a quasiorder).
- If $A \leq_m^P B$ and $B \in \mathrm{P}$, then $A \in \mathrm{P}$.
- If $A \leq_m^P B$ and $B \in \mathrm{NP}$, then $A \in \mathrm{NP}$.

NP-completeness

Definition 50

- Language B is NP-hard if every problem A in NP is polynomial time reducible to B.
- An NP-hard language B which belongs to NP is called NP-complete.

- If we want to show that problem B is NP-complete we can
  1. show that $B \in \mathrm{NP}$ and
  2. find another NP-complete problem A and reduce it to B (show that $A \leq_m^P B$).

Assuming $\mathrm{P} \neq \mathrm{NP}$, if B is an NP-complete problem then $B \notin \mathrm{P}$.

An NP-complete problem

Tiling

Instance: Set of colors B, natural number s, square grid S of size $s \times s$, in which border cells have outer edges colored by colors in B. Set of tile types K, every tile is a square with edges colored by colors in B.

Question: Is it possible to place tiles from K to the cells of S without rotation, so that the tiles sharing a border have matching color and the tiles placed in a border cell have the colors matching outer edge colors of S.

Theorem 51

TILING is NP-complete.

Satisfiability

Literal	a variable (e.g. x) or its negation (e.g. $\overline{x}$).
Clause	a disjunction of literals.
Conjunctive normal form (CNF)	a formula is in CNF if it is a conjunction of clauses.

Satisfiability (SAT)

Instance: A formula φ in CNF.

Question: Is there an assignment v of truth values to variables so that $\varphi(v)$ is satisfied?

Theorem 52 (Cook-Levin theorem)

SAT belongs to P *if and only if* $\mathrm{P} = \mathrm{NP}$*. In particular SAT is* NP*-complete.*

3-Satisfiability

3-CNF A formula φ is in 3-CNF if it is in CNF and every clause consists of exactly 3 literals.

3-Satisfiability (3-SAT)	
Instance:	Formula φ in 3-CNF.
Question:	Is there an assignment v of truth values to variables so that $\varphi(v)$ is satisfied?

Theorem 53

3-SATISFIABILITY is NP-complete.

Vertex Cover

Vertex Cover

Instance:	An undirected graph $G = (V, E)$ and an integer $k \geq 0$.
Question:	Is there a set of vertices $S \subseteq V$ of size at most k so that each edge $\{u, v\} \in E$ has one of its endpoints in S (that is $\{u, v\} \cap S \neq \emptyset$)?

Theorem 54 (Without proof)

VERTEX COVER is NP-complete.

Vertex Cover (related problems)

- NP-complete problems related to VERTEX COVER:
  - CLIQUE: Does a given graph G contain a complete subgraph (=clique) on k vertices?
  - INDEPENDENT SET: Does a given graph G contain an independent set of size k? (A set of vertices is independent in G, if it induces subgraph without edges.)
- An analogous problem EDGE COVER, in which we are looking for a smallest set of edges which together contain all vertices, is solvable in polynomial time.

Hamiltonian Cycle

Hamiltonian Cycle (HC)

Instance: An undirected graph $G = (V, E)$.

Question: Is there a cycle in G which would go through all vertices?

Theorem 55 (Without proof)

Hamiltonian cycle is an NP-complete problem.

Travelling Salesperson

Traveling Salespersion (TSP)

Instance: A set of cities $C = \{c_1, \ldots, c_n\}$, distances $d(c_i, c_j) \in \mathbb{N}$ between all pairs of cities, a limit $D \in \mathbb{N}$.

Question: Is there a permutation of cities $c_{\pi(1)}, c_{\pi(2)}, \ldots, c_{\pi(n)}$, which satisfies

$$\left(\sum_{i=1}^{n-1} d(c_{\pi(i)}, c_{\pi(i+1)})\right) + d(c_{\pi(n)}, c_{\pi(1)}) \leq D?$$

Theorem 56

TRAVELLING SALESPERSON is an NP-complete problem.

3-Dimensional Matching

3-Dimensional Matching (3DM)

Instance: Set $M \subseteq W \times X \times Y$, where W, X, and Y are sets of size q.

Question: Can we find a perfect matching in M? In particular, is there a set $M' \subseteq M$ of size q so that all triples in M' are pairwise disjoint?

Theorem 57 (Without proof)

3-Dimensional Matching is an NP*-complete problem.*

Partition

Partition

Instance: A set of items A and a natural number $s(a)$ associated with each item $a \in A$ (weight, value, size).

Question: Is there a subset $A' \subseteq A$ satisfying

$$\sum_{a \in A'} s(a) = \sum_{a \in A \setminus A'} s(a)?$$

Theorem 58

PARTITION is an NP-complete problem.

Knapsack

Knapsack

Instance: A set of items A and for each item a we have specified its size $s(a) \in \mathbb{N}$ and value $v(a) \in \mathbb{N}$. The knapsack size $B \in \mathbb{N}$ and value limit $K \in \mathbb{N}$.

Question: Is there a subset of items $A' \subseteq A$ satisfying

$$\sum_{a \in A'} s(a) \le B \text{ and } \sum_{a \in A'} v(a) \ge K?$$

Theorem 59

KNAPSACK is an NP-complete problem.

- A simple reduction from PARTITION.

Scheduling

Scheduling

Instance:	A set of tasks U, processing time $d(u) \in \mathbb{N}$ associated with every task $u \in U$, number of processors m, deadline $D \in \mathbb{N}$.
Question:	Is it possible to assign all tasks to processors so that the (parallel) processing time is at most D?

Theorem 60

SCHEDULING is an NP-complete problem.

- A simple reduction from PARTITION.

Pseudopolynomial algorithms and strong NP-completeness

Knapsack (optimization version)

Knapsack

Instance:	Set of items A, size $s(a) \in \mathbb{N}$ and value $v(a) \in \mathbb{N}$ associated with each item $a \in A$. Size of the knapsack $B \in \mathbb{N}$.
Feasible solution:	Set $A' \subseteq A$ satisfying that $\sum_{a \in A'} s(a) \leq B$
Goal:	Maximize sum of values of items in A', that is $\sum_{a \in A'} v(a)$.

Pseudopolynomial algorithm for Knapsack (1)

Input: Knapsack size B, number of items n. Array of sizes s and array of values v (both of size n). We assume that $(\forall i)[0 \leq s(i) \leq B]$.

Output: Set of items A' with total size of items at most B and with maximum total value.

```
1: $V \leftarrow \sum_{i=1}^{n} v[i]$
2: $T$ is a new matrix with dimensions $(n+1) \times (V+1)$, where
   $T[j,c]$ in the end contains a set of items chosen from
   $\{1, \ldots, j\}$ with total value $c$ and the minimum total size of
   items.
3: $S$ is a new matrix with dimensions $(n+1) \times (V+1)$, where
   $S[j,c]$ in the end contains the sum of sizes of items in set
   $T[j,c]$ or $B+1$, if no set is assigned to $T[j,c]$.
```

Pseudopolynomial algorithm for Knapsack (2)

```
 4: $T[0,0] \leftarrow \emptyset$, $S[0,0] \leftarrow 0$
 5: for $c \leftarrow 1$ to $V$ do
 6:     $T[0,c] \leftarrow \emptyset$, $S[0,c] \leftarrow B+1$
 7: end for
 8: for $j \leftarrow 1$ to $n$ do
 9:     $T[j,0] \leftarrow \emptyset$, $S[j,0] \leftarrow 0$
10:     for $c \leftarrow 1$ to $V$ do
11:         $T[j,c] \leftarrow T[j-1,c]$, $S[j,c] \leftarrow S[j-1,c]$
12:         if $v[j] \leq c$ and $S[j,c] > S[j-1,c-v[j]] + s[j]$ then
13:             $T[j,c] \leftarrow T[j-1,c-v[j]] \cup \{j\}$
14:             $S[j,c] \leftarrow S[j-1,c-v[j]] + s[j]$
15:         end if
16:     end for
17: end for
18: $c \leftarrow \max\{c' \mid S[n,c'] \leq B\}$
19: return $T[n,c]$
```

Pseudopolynomial algorithm for Knapsack (3)

- The described algorithm works in time $\Theta(nV)$ (if we consider arithmetic operations as constant time).
- In general, the algorithm does not work in polynomial time because the size of the input is $O(n \log_2(B + V))$.
- Algorithms of this kind shall be called pseudopolynomial.

Number problems

Definition 61

Let A be a decision problem and let I be an instance of A. Then

$\mathrm{len}(I)$	denotes the length (=number of bits) of encoding of I when using binary encoding of numbers.
$\max(I)$	denotes the value of a maximum number parameter in I.

We say that A is a number problem, if for any polynomial p there is an instance I of A with $\max(I) > p(\mathrm{len}(I))$.

For instance

- KNAPSACK and PARTITION are number problems.
- SATISFIABILITY and TILING are not number problems.

Pseudopolynomial algorithm

Definition 62

We say that an algorithm which solves problem A is pseudopolynomial if its running time is bounded by a polynomial in two variables $\operatorname{len}(I)$ and $\max(I)$.

- We usually measure complexity of an algorithm only with respect to $\operatorname{len}(I)$.
- If for some polynomial p and for every instance I of A we have that $\max(I) \leq p(\operatorname{len}(I))$ then a pseudopolynomial algorithm is actually polynomial.
- Also, if the numbers in I would be encoded in unary, a pseudopolynomial algorithm would run in polynomial time.

Examples of Pseudopolynomial Algorithms

- Sieve of Eratosthenes
- Naive factorization
- Counting sort

Strong NP-completeness

Definition 63

- Let A be a decision problem and let p be a polynomial. Then $A(p)$ denotes the restriction of problem A to instances I which satisfy $\max(I) \leq p(\operatorname{len}(I))$.
- We say that problem A is strongly NP-complete, if there is a polynomial p for which $A(p)$ is NP-complete.

- Any NP-complete problem which is not a number problem is strongly NP-complete.
- If there is a strongly NP-complete problem which can be solved by a pseudopolynomial algorithm then $\mathrm{P} = \mathrm{NP}$.

Binary vs. unary encoding

- Pseudopolynomial=polynomial when considering unary encoding.
- Strongly NP-complete=NP-complete even when considering unary encoding.

Binary encoding	Unary encoding
P	Solvable by a pseudopolynomial algorithms.
NP-complete	Strongly NP-complete.

Strong NP-completeness of TSP

Traveling Salesperson (TSP)

Instance: A set of cities $C = \{c_1, \ldots, c_n\}$, distances $d(c_i, c_j) \in \mathbb{N}$ between all pairs of cities, a limit $D \in \mathbb{N}$.

Question: Is there a permutation of cities $c_{\pi(1)}, c_{\pi(2)}, \ldots, c_{\pi(n)}$, which satisfies

$$\left(\sum_{i=1}^{n-1} d(c_{\pi(i)}, c_{\pi(i+1)})\right) + d(c_{\pi(n)}, c_{\pi(1)}) \leq D?$$

Theorem 64

Travelling Salesperson is a strongly NP-complete problem.

Approximation algorithms

Optimization problem

Definition 65

- We define optimization problem as a triple $A = (D_A, S_A, \mu_A)$, where
  - $D_A \subseteq \Sigma^*$ is a set of instances,
  - $S_A(I)$ assigns a set of feasible solutions to each $I \in D_A$
  - $\mu_A(I, \sigma)$ assigns a positive rational value to every $I \in D_A$ and every feasible solution $\sigma \in S_A(I)$.
- If A is a maximization problem, then an optimum solution to instance I is a feasible solution $\sigma \in S_A(I)$, which has the maximum value $\mu_A(I, \sigma)$.
- If A is a minimization problem, then an optimum solution to instance I is a feasible solution $\sigma \in S_A(I)$, which has the minimum value $\mu_A(I, \sigma)$.
- The value of an optimum solution is denoted $\mathrm{opt}(I)$.

Bin Packing

Bin Packing (BP)

Instance:	Set of items U, a size $s(u)$ associated with each item u. The size is a rational value from interval $[0, 1]$.
Feasible solution:	Splitting of items to pairwise disjoint bins $U_1, \ldots, U_m$, satisfying $$(\forall i \in \{1, \ldots, m\}) \left[\sum_{u \in U_i} s(u) \leq 1 \right].$$
Goal:	Minimize the number of bins m.

The decision version is equivalent to SCHEDULING.

Approximation algorithm

Definition 66

Algorithm R is called approximation algorithm for optimization problem A, if for each instance $I \in D_A$ the output of $R(I)$ is a feasible solution $\sigma \in S_A(I)$ (if there is any).

- If A is a maximization problem, then $\varepsilon \geq 1$ is an approximation ratio of algorithm R, if for all instances $I \in D_A$ we have that $\mathrm{opt}(I) \leq \varepsilon \cdot \mu_A(I, R(I))$.
- If A is a minimization problem, then $\varepsilon \geq 1$ is an approximation ratio of algorithm R, if for all instances $I \in D_A$ we have that $\mu_A(I, R(I)) \leq \varepsilon \cdot \mathrm{opt}(I)$.

An approximation algorithm for Bin Packing

Algorithm 1 First Fit (FF)

```
1: Take items as they come and for each item try to find a bin
   in which it fits.
2: If no such bin exists, add a new bin with the item in it.
```

Theorem 67

- *If I is an instance of* Bin Packing *and if m is the number of bins created by algorithm FF on instance I, then* $m < 2 \cdot \mathrm{opt}(I)$.
- *For any m there is an instance I such that $\mathrm{opt}(I) \geq m$ for which FF returns a solution with at least $\frac{5}{3}\mathrm{opt}(I)$ bins.*

A better algorithm for Bin Packing

Algorithm 2 First Fit Decreasing (FFD)

```
1: Sort the items by their value decreasing.
2: Take items from the biggest to smallest and for each item try
   to find a bin in which it fits.
3: If no such bin exists, add a new bin with the item in it.
```

Theorem 68 (Without proof)

- *If I is an instance of* BIN PACKING *and if m is the number of bins produced by algorithm FFD on instance I, then $m \leq \frac{11}{9} \cdot \mathrm{opt}(I) + 4$.*
- *For each m there is an instance I, such that $\mathrm{opt}(I) \geq m$, for which algorithm FFD produces at least $\frac{11}{9}\mathrm{opt}(I)$ bins.*

Travelling Salesperson (optimization version)

Traveling Salesperson (TSP)

Instance:	Set of cities $C = \{c_1, \dots, c_n\}$, distances $d(c_i, c_j) \in \mathbb{N}$ between all pairs of cities.
Feasible solution:	Permutation of cities $c_{\pi(1)}, c_{\pi(2)}, \dots, c_{\pi(n)}$.
Goal:	Minimize $$\left(\sum_{i=1}^{n-1} d(c_{\pi(i)}, c_{\pi(i+1)})\right) + d(c_{\pi(n)}, c_{\pi(1)}).$$

Theorem 69

Travelling Salesperson is an NP-complete problem.

Hardness of approximation

Theorem 70

if $\mathrm{P} \neq \mathrm{NP}$, there is no polynomial approximation algorithm with a constant approximation ratio for TRAVELLING SALESPERSON.

- There is a $\frac{3}{2}$-approximation algorithm for TSP if the distance function satisfies triangle inequality.
- There is a polynomial approximation scheme for TSP in Euclidean plane.

Approximation scheme for Knapsack

Input: Knapsack size B, number of items n. Array of sizes s and array of values v (both of size n). We assume that $(\forall i)[0 \le s(i) \le B]$. Rational number $\varepsilon > 0$.

Output: Set of items A' with total size of items at most B and with total value at least $\frac{1}{1+\varepsilon}\mathrm{opt}(I)$.

```
 1: function BAPX($I = (B, n, s, v), \varepsilon$)
 2:     $m \leftarrow \arg\max_{1 \le i \le n} v[i]$
 3:     if $\varepsilon \ge n - 1$ then return $\{m\}$
 4:     end if
 5:     $t \leftarrow \left\lfloor \log_2\left(\frac{\varepsilon \cdot v[m]}{n}\right) \right\rfloor - 1$
 6:     $c$ is a new array of size $n$
 7:     for $i \leftarrow 1$ to $n$ do
 8:         $c[i] \leftarrow \left\lfloor \frac{v[i]}{2^t} \right\rfloor$
 9:     end for
10:     Using pseudopolynomial algorithm for KNAPSACK find an
        optimum solution to instance $B, s, c$ and return the solution.
11: end function
```

Properties of Knapsack approximation

Theorem 71

Let I be an instance of KNAPSACK *and let $\varepsilon > 0$ be a rational number.*

- *Let $\mathrm{bapx}(I, \varepsilon)$ be a value of solution returned by algorithm BAPX for a given instance I and rational number $\varepsilon > 0$, then*
$$\mathrm{opt}(I) \leq (1 + \varepsilon) \cdot \mathrm{bapx}(I, \varepsilon).$$
- *Algorithm BAPX works in time $O(\frac{1}{\varepsilon} n^3)$ (if we consider arithmetic operations as constant time).*

Fully polynomial time approximation scheme

Definition 72

- Algorithm ALG is an approximation scheme for an optimization problem A, if on the input instance $I \in D_A$ and a rational number $\varepsilon > 0$ it returns a solution $\sigma \in S_A(I)$ with approximation ratio $1 + \varepsilon$.
- If ALG works in polynomial time with respect to $\mathrm{len}(I)$, then it is a polynomial time approximation scheme.
- If ALG works in polynomial time with respect to both $\mathrm{len}(I)$ and $\frac{1}{\varepsilon}$, it is a fully polynomial time approximation scheme (FPTAS).

- BAPX is a fully polynomial time approximation scheme for KNAPSACK.

FPTAS and strong NP-completeness

Theorem 73

Let A be an optimization problem and let us assume that for any instance $I \in D_A$ the value $\mu_A(I, \sigma) \in \mathbb{N}$. Let us assume that there is a polynomial q of two variables so that for any instance $I \in D_A$ we have that

$$\mathrm{opt}(I) < q(\mathrm{len}(I), \max(I)).$$

If there is a fully polynomial time approximation scheme for A, then there is also a pseudopolynomial algorithm for A.

- If $\mathrm{P} \neq \mathrm{NP}$, there is no FPTAS for any strong NP-complete problem satisfying the assumptions of above theorem.

Classes co-NP and #P

Unsatisfiability

Unsatisfiability (UNSAT)	
Instance:	Formula φ in CNF
Question:	Is it true, that for any assignment v of values to variables $\varphi(v) = 0$ (unsatisfied)?

- We do not know a polynomial time verifier for problem UNSAT, this problem most probably does not belong to class NP.
- Language UNSAT is (more or less) the complement of language SAT, because for any formula φ in CNF we have

$$\varphi \in \text{UNSAT} \iff \varphi \notin \text{SAT}$$

Class co-NP

Definition 74

We say that language A belongs to the class $\mathrm{co\text{-}NP}$ if and only if its complement $\overline{A}$ belongs to the class NP.

- For instance UNSAT belongs to $\mathrm{co\text{-}NP}$. (It is easy to recognize languages which do not encode a formula.)
- Language L belongs to $\mathrm{co\text{-}NP}$, iff there is a polynomial time verifier V which satisfies that

$$L = \{x \mid (\forall y)\,[V(x,y) \text{ accepts }]\} .$$

- We have that $\mathrm{P} \subseteq \mathrm{NP} \cap \mathrm{co\text{-}NP}$.

co-NP-completeness

Definition 75

Problem A is co-NP-complete, if

1. A belongs to class co-NP and
2. every problem $B \in$ co-NP is polynomial time reducible to A.

- Language A is co-NP-complete, if and only if complement $\overline{A}$ is NP-complete.
- For example UNSAT is an co-NP-complete problem.
- If there is an NP-complete language A, which belongs to co-NP, then NP $=$ co-NP.

Class #P

Definition 76

Function $f : \Sigma^* \mapsto \mathbb{N}$ belongs to class $\#\mathrm{P}$, if there is a polynomial time verifier V such that for each $x \in \Sigma^*$

$$f(x) = |\{y \mid V(x, y) \text{ accepts}\}|.$$

- We can associate a function $\#A$ in $\#\mathrm{P}$ with every problem $A \in \mathrm{NP}$ (given by the “natural” polynomial time verifier for A).
- Natural verifier verifies that y is a solution to the search problem corresponding to A.
- For example the natural verifier for SAT accepts a pair φ, v, if φ is a CNF and v is a satisfying assignment for φ.
- Then $\#\text{SAT}(\varphi) = |\{v \mid \varphi(v) = 1\}|$.

Class #P (properties)

Consider function $f \in \#\mathrm{P}$ and problem:

Nonzero Value of f
Instance: $x \in \Sigma^*$.
Question: $f(x) > 0$?

- Problem Nonzero Value of f belongs to NP.
- Value of $f \in \#\mathrm{P}$ can be obtained by using polynomial number of queries about an element belonging to the set $\{(x, N) \mid f(x) \geq N\}$.
- Value of $f \in \#\mathrm{P}$ can be computed in polynomial space.

Reducing a function to another function

Definition 77

Function $f : \Sigma^* \mapsto \mathbb{N}$ is polynomial time reducible to function $g : \Sigma^* \mapsto \mathbb{N}$ ($f \leq_P g$) if there are functions $\alpha : \Sigma^* \times \mathbb{N} \mapsto \mathbb{N}$ a $\beta : \Sigma^* \mapsto \Sigma^*$, which can be computed in polynomial time and

$$(\forall x \in \Sigma^*)\left[f(x) = \alpha\left(x, g\left(\beta(x)\right)\right)\right]$$

- This corresponds to the fact that f can be computed in polynomial time with one call of function g (if this call is a constant time operation).

Parsimonious reduction

Definition 78

We say that problem $A \in \Sigma^*$ is polynomial time reducible to problem $B \in \Sigma^*$ by parsimonious reduction ($A \leq_c^P B$), if there is a function $f : \Sigma^* \mapsto \Sigma^*$ computable in polynomial time such that

$$|\{y \mid V_A(x, y) \text{ accepts}\}| = |\{y \mid V_B(f(x), y) \text{ accepts}\}|,$$

where V_A and V_B are natural verifiers for A and B.

- If $A \leq_c^P B$, then $\#A \leq_P \#B$.
- The reductions we have presented during the lecture can be modified into parsimonious reductions.

#P-completeness

Definition 79

We say that function $f : \Sigma^* \mapsto \mathbb{N}$ is #P-complete, if

1. $f \in \#\mathrm{P}$ and
2. every function $g \in \#\mathrm{P}$ is polynomial time reducible to f.

- For example #SAT, #VERTEX COVER and other counting versions of NP-complete problems are #P-complete.
- Using just parsimonious reductions.
- There are problems in P such that their counting versions are #P-complete.

Number of perfect matchings in a bipartite graph

Perfect matching in a bipartite graph (BPM)

Instance:	Bipartite graph $G = (V = A \cup B, E \subseteq A \times B)$, where $\lvert A\rvert = \lvert B\rvert$.
Question:	Is there a matching in G of size $\lvert A\rvert = \lvert B\rvert$?

Theorem 80 (Without proof)

Function #BPM is $\#\mathrm{P}$-complete.

Permanent of a matrix

Definition 81

Let A be a matrix of type $n \times n$. Then we define permanent of A as

$$\operatorname{perm}(A) = \sum_{\pi \in S(n)} \prod_{i=1}^{n} a_{i,\pi(i)},$$

where $S(n)$ is a set of permutations over set $\{1, \dots, n\}$.

- Like “determinant” without a sign of permutation.
- If A is a adjacency matrix of a bipartite graph G, then $\operatorname{perm}(A)$ computes the number of perfect matchings of G.

Theorem 82 (Without proof)

Function perm *is* $\#\mathrm{P}$*-complete.*

#DNF-SAT

Term is a conjunction of literals.

Disjunctive normal form (DNF) is a disjunction of terms.

DNF-Satisfiability (DNF-SAT)	
Instance:	Formula φ in DNF
Question:	Is there an assignment v such that $\varphi(v)$ is satisfied?

- DNF-SAT is decidable in polynomial time.
- Function #DNF-SAT is #P-complete.

An Advertisement

For those who want to know more, I can recommend lectures in summer semester:

Computability (NTIN064)

Lectured by doc. RNDr. Antonín Kučera, CSc.

Complexity (NTIN063)

Lectured by doc. RNDr. Ondřej Čepek, Ph.D.