Chapter 1

Introduction

1.1 Basic notions

1.1.1 Algorithm. By an *algorithm*, we mean a well defined finite procedure, i.e. a sequence of steps, such that for a given input (values of an instance) it produces an output (solution).

1.1.2 Problem. A *problem* is a general specification of a relationship input/solution. By an *instance* of a problem $\mathcal{U}$, we mean an assignment values of all parameters of which the problem consists. In other words, an instance of a problem is a correct example of assignment.

1.1.3 We say that an algorithm $\mathcal{A}$ *solves* a problem $\mathcal{U}$ if for every input (every instance of $\mathcal{U}$) it gives a correct solution.

Note that the above statement means that every algorithm which solves a problem must terminate for every input. It means that an algorithm which does not terminate for an input cannot solve any problem.

1.1.4 Time complexity. There are two different ways to measure time complexity of algorithms.

1. Worst case analysis. It is the asymptotic estimation $T(n)$ of the upper bound for the length of time that is needed for any instance of size n.
2. Average complexity. It is the asymptotic estimation $T_{aver}(n)$ of the average length of time that is needed to solve an instance of size n.

1.2 Asymptotic growth of functions

1.2.1 Symbol $\mathcal{O}$. Given a nonnegative function $g(n)$, we say that a nonnegative function $f(n)$ *is* $\mathcal{O}(g(n))$ if there exist a positive constant c and a natural number n_0 such that

$$f(n) \leq c\,g(n) \quad \text{for all } n \geq n_0.$$

We can consider $\mathcal{O}(g(n))$ to be the class of all nonnegative functions $f(n)$:

$$\mathcal{O}(g(n)) = \{f(n) \mid \exists c > 0, n_0 \text{ such that } f(n) \leq c\,g(n) \ \ \forall n \geq n_0\}.$$

1.2.2 Symbol Ω. Given a nonnegative function $g(n)$, we say that a nonnegative function $f(n)$ *is* $\Omega(g(n))$ if there exists a positive constant c and a natural number n_0 such that

$$f(n) \geq c\,g(n) \quad \text{for all } n \geq n_0.$$

We can consider $\Omega(g(n))$ to be the class of all nonnegative functions $f(n)$:

$$\Omega(g(n)) = \{f(n) \mid \exists c > 0, n_0 \text{ such that } f(n) \geq c\,g(n) \ \ \forall\, n \geq n_0\}.$$

1.2.3 Remark. It holds that a function $f(n)$ is $\Omega(g(n))$ iff the function $g(n)$ is $\mathcal{O}(f(n))$.

1.2.4 Symbol Θ. Given a nonnegative function $g(n)$, we say that a non negative function $f(n)$ *is* $\Theta(g(n))$ if there exists positive constants c_1, c_2 and a natural number n_0 such that

$$c_1\, g(n) \le f(n) \le c_2\, g(n) \quad \text{for all } n \ge n_0.$$

We can consider $\Theta(g(n))$ to be the class of all nonnegative functions $f(n)$:

$$\Theta(g(n)) = \{f(n) \mid \exists c_1, c_2 > 0, n_0 \text{ such that } c_1\, g(n) \le f(n) \le c_2\, g(n) \;\; \forall\, n \ge n_0\}.$$

1.2.5 Remark. We have $f(n)$ is $\Theta(g(n))$ if and only if $f(n)$ is both $\mathcal{O}(g(n))$ and $\Omega(g(n))$.

1.2.6 Symbol small o. Given a nonnegative function $g(n)$. We say that a nonnegative function $f(n)$ *is* $o(g(n))$ if for every positive constant c there exists a natural number n_0 such that

$$0 \le f(n) < c\, g(n) \quad \text{for all } n \ge n_0.$$

We can consider $o(g(n))$ to be the class of all nonnegative functions $f(n)$:

$$o(g(n)) = \{f(n) \mid \forall\, c > 0\; \exists\, n_0 \text{ such that } 0 \le f(n) < c\, g(n) \;\; \forall\, n > n_0\}.$$

1.2.7 Remark. A nonnegative function $f(n)$ is $\mathcal{O}(g(n))$ roughly means that the function $f(n)$ does not grow asymptotically more than $g(n)$. On the other hand, to say that a nonnegative function $f(n)$ is $o(g(n))$ roughly means that the function $f(n)$ grows asymptotically less than the function $g(n)$.

1.2.8 Symbol small ω. Given a nonnegative function $g(n)$, we say that a nonnegative function $f(n)$ *is* $\omega(g(n))$ if for every positive constant c there exists a natural number n_0 such that

$$0 \le c\, g(n) < f(n) \quad \text{for all } n \ge n_0.$$

We can consider $\omega(g(n))$ to be a class of all nonnegative functions $f(n)$:

$$\omega(g(n)) = \{f(n) \mid \forall\, c > 0 \;\; \text{there is } n_0 \text{ such that } 0 \le c\, g(n) < f(n) \;\; \forall\, n > n_0\}.$$

1.2.9 Remark. Roughly speaking, we say that a nonnegative function $f(n)$ is $\Omega(g(n))$ means that the function $f(n)$ grows asymptotically at least as the function $g(n)$. On the other hand, $f(n)$ is $\omega(g(n))$ means roughly that the function $f(n)$ grows asymptotically more than the function $g(n)$.

1.2.10 Notation. Symbols $\mathcal{O}, \Omega, \Theta, o, \omega$ represent classes of functions so we will write $f(n) \in \mathcal{O}(g(n))$; similarly for other symbols $\Omega.\Theta, o, \omega$.

1.2.11 Proposition. Given two nonnegative functions $f(n)$ and $g(n)$, then

- $f(n) \in o(g(n))$ if and only if $\lim_{n\to\infty} \frac{f(n)}{g(n)} \in 0$;
- $f(n) \in \omega(g(n))$ if and only if $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$.

Justification. Let us write what it meas that $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$:

$$\forall\, \varepsilon > 0\; \exists\, n_0 \in \mathbb{N} \;\text{ such that }\; \forall\, n \ge n_0 \;\text{ it holds }\; |\frac{f(n)}{g(n)}| < \varepsilon.$$

The fact $|\frac{f(n)}{g(n)}| < \varepsilon$ can be rewritten as $f(n) < \varepsilon\, g(n)$. Denote $c := \varepsilon$, than we get that $f(n)$ is in $o(g(n))$.

3) Analogously, $\lim_{n\to\infty}\frac{f(n)}{g(n)} = a$, $a > 0$, means that

$$\forall\,\varepsilon > 0\ \exists\, n_0 \in \mathbb{N} \text{ such that } \forall\, n \geq n_0 \text{ it holds } |\frac{f(n)}{g(n)} - a| < \varepsilon.$$

Equivalently, $(a-\varepsilon)g(n) < f(n) < (a+\varepsilon)g(n)$. Choose $\varepsilon = \frac{a}{2}$, we get

$$\frac{a}{2}\, g(n) < f(n) < \frac{3a}{2}\, g(n);$$

hence $f(n)$ is $\Theta(g(n))$.

1.2.12 Transitivity. Given three nonnegative functions $f(n)$, $g(n)$ and $h(n)$.

1. If $f(n) \in \mathcal{O}(g(n))$ and $g(n) \in \mathcal{O}(h(n))$, then $f(n) \in \mathcal{O}(h(n))$.
2. If $f(n) \in \Omega(g(n))$ and $g(n) \in \Omega(h(n))$, then $f(n) \in \Omega(h(n))$.
3. If $f(n) \in \Theta(g(n))$ and $g(n) \in \Theta(h(n))$, then $f(n) \in \Theta(h(n))$.

1.2.13 Reflexivity. For all nonnegative functions $f(n)$, we have: $f(n) \in \mathcal{O}(f(n))$, $f(n) \in \Omega(f(n))$ and $f(n) \in \Theta(f(n))$.

1.2.14 Proposition. $f(n) \in \Theta(g(n))$ if and only if $g(n) \in \Theta(f(n))$.

1.2.15 Examples.

1. For every $a > 1$ and $b > 1$, we have
$$\log_a(n) \in \Theta(\log_b(n)).$$
2. It holds that
$$\lg n! \in \Theta(n \lg n).$$

The second part of the above example follows from the following theorem.

1.2.16 Theorem (Gauss). For every $n \geq 1$

$$n^{\frac{n}{2}} \leq n! \leq \left(\frac{n+1}{2}\right)^n.$$

Justification. We will use the fact that for every two positive numbers a, b it holds that $\frac{a+b}{2} \geq \sqrt{ab}$.

Let us write $(n!)^2$ as

$$(n!)^2 = n\,(n-1)\,\ldots 2\,1\,1\,2\,\ldots\,(n-1)\,n = \prod_{i=1}^{n}(n-i+1)i.$$

Therefore

$$n! = \prod_{i=1}^{n}\sqrt{(n-i+1)i} \leq \prod_{i=1}^{n}\frac{n+1}{2} = \left(\frac{n+1}{2}\right)^n,$$

since for every i we have $\sqrt{(n-i+1)i} \leq \frac{n-i+1+i}{2}$. We have shown the first estimate.

On the other hand, for every i we have $n \leq (n-i+1)i$, hence $n^n \leq (n!)^2$. Since the both expressions are positive, we can form their square root and get $n^{\frac{n}{2}} \leq n!$.

1.2.17 Theorem. Given a non negative function $f(n)$ which is non decreasing. If $f(\frac{n}{2}) \in \Theta(f(n))$, then

$$\sum_{i=1}^{n} f(i) \in \Theta(n\, f(n)).$$

Justification. The fact that $\sum_{i=1}^{n} f(i) \in \mathcal{O}(n\, f(n))$ is clear: f is non decreasing.

Further, there is a positive constant c such that for sufficiently big n we have $c\, f(n) \le f(\frac{n}{2})$. Hence

$$\sum_{i=1}^{n} f(i) \ge f(\frac{n}{2}) + \ldots + f(n) \ge \frac{n}{2}\, c\, f(n).$$

This means that $\sum_{i=1}^{n} f(i) \ge \frac{c}{2}\, n\, f(n)$ and therefore $\sum_{i-1}^{n} f(i) \in \Omega(n\, f(n))$.

1.2.18 Remark. The property of the theorem above has for example $f(n) = n^d$ for a natural number $d \ge 1$. On the other hand the function $f(n) = 2^n$ does not fulfill it. To find an asymptotic growth of $\sum_{i=1}^{n} 2^i$ the following method can be used:

Using mathematical induction we show that there is a constant $c > 0$ such that

$$\sum_{i=1}^{n} 2^i \le c\, 2^n.$$

1. Basic step. We know that $\sum_{i=1}^{1} 2^i = 2$ and that $2 \le c\, 2$ for every constant $c \ge 1$.
2. Induction step. Assume that $\sum_{i=1}^{n} 2^i \le c\, 2^n$. Then

$$\sum_{i=1}^{n+1} 2^i = \sum_{i=1}^{n} 2^i + 2^{n+1} \le c\, 2^n + 2^{n+1} = \left(\frac{1}{2} + \frac{1}{c}\right) c\, 2^{n+1}.$$

Now, to finish the proof it suffices to find c such that $\frac{1}{2} + \frac{1}{c} \le 1$. And this is equivalent with the condition that $c \ge 2$.

1.2.19 There is another way how to deal with finding an asymptotic growth of functions of the type

$$\sum_{i=1}^{n} f(i).$$

We show it only for non decreasing functions $f(n)$ (for non increasing you have to change the inequalities):

$$\int_0^n f(x)\, dx \le \sum_{i=1}^{n} f(i) \le \int_1^{n+1} f(x)\, dx.$$

If the integral is improper, it is useful to look for an estimate for $\sum_{i=2}^{n} f(i)$ only.

1.3 Solving Recurrences

1.3.1 Theorem — Master Theorem. Given natural numbers $a \ge 1$, $b > 1$ and a function $f(n)$, a recurrence $T(n)$ is defined for all natural numbers by with

$$T(n) = a\, T(\frac{n}{b}) + f(n),$$

where $\frac{n}{b}$ means either $\lfloor \frac{n}{b} \rfloor$ or $\lceil \frac{n}{b} \rceil$.

1. If $f(n) \in \mathcal{O}(n^{\log_b a - \varepsilon})$ for a constant $\varepsilon > 0$, then $T(n) \in \Theta(n^{\log_b a})$.
2. If $f(n) \in \Theta(n^{\log_b a})$, then $T(n) \in \Theta(n^{\log_b a}\ \lg n)$.
3. If $f(n) \in \Omega(n^{\log_b a + \varepsilon})$ for a constant $\varepsilon > 0$ and $a\, f(\frac{n}{b}) \le c\, f(n)$ for a constant $c < 1$ given sufficiently large n, then $T(n) \in \Theta(f(n))$.

1.3.2 Remark. There are some examples that are not solved by the Master Theorem:

1. We have $f(n) \in \mathcal{O}(n^{\log_b a})$ but there is no $\varepsilon > 0$ so that $f(n) \in \mathcal{O}(n^{\log_b a - \varepsilon})$. In other words, $f(n)$ is not polynomially smaller than $\mathcal{O}(n^{\log_b a})$.
2. We have $f(n) \in \Omega(n^{\log_b a})$ but there is no $\varepsilon > 0$ so that $f(n) = \mathcal{O}(n^{\log_b a + \varepsilon})$ (in other words, $f(n)$ is not polynomially greater than $\mathcal{O}(n^{\log_b a})$) or there is not constant $c < 1$ for which $a\, f(\frac{n}{b}) \le c\, f(n)$.

1.3.3 Proposition. If $f(n) \in \Theta(n^{\log_b a} \lg^k n)$ for $k \ge 0$, then for $T(n)$ given by

$$T(n) = a\, T(\frac{n}{b}) + f(n),$$

we have: $T(n) \in \Theta(n^{\log_b a} \lg^{k+1} n)$.

1.3.4 Solving recurrences using recursion trees. There is another method how to solve recurrences. We show it on the two following examples. The first one is also solvable by Master theorem, to the second one the Master theorem is not applicable.

1.3.5 Example 1. Solve the following recurrence

$$T(n) = 3T\left(\frac{n}{4}\right) + n^2.$$

Solution: We form the tree by levels so that it describes the recursion of $T(n)$. In the 0 level for a computation of $T(n)$ we need n^2 steps necessary to compute $T(n)$ (if we already have $T\left(\frac{n}{4}\right)$).

In the first level the computation of $T(n)$ breaks down to three computations $T(\frac{n}{4})$. For this we need $3 \cdot (\frac{n}{4})^2 = \frac{3}{16}\, n^2$.

When we go from level i to level $i+1$ every node has three successors and each needs one sixteenth of the previous one. Therefor the sum in the i level is equal to $(\frac{3}{16})^i\, n^2$.

The last level has nodes denoted by $T(1)$ and it is the end of recursion. The number of levels corresponds to $\lceil \log_4 n \rceil$. Moreover, in the last level there is $3^{\log_4 n} = n^{\log_4 3}$ values $T(1)$. Hence we have

$$T(n) = \sum_{i=0}^{\lceil \log_4 n \rceil} \left(\frac{3}{16}\right)^i n^2 \, + \, \Theta(n^{\log_4 3}).$$

Thus

$$T(n) < n^2 \sum_{i=0}^{\infty} \left(\frac{3}{16}\right)^i \, + \, \Theta(n^{\log_4 3}) = n^2 \frac{1}{1 - \frac{3}{16}} + \Theta(n^{\log_4 3}) = \frac{16}{13} n^2 + \Theta(n^{\log_4 3}).$$

We have got $T(n) \in \Theta(n^2)$.

1.3.6 Example 2. Solve

$$T(n) = T\left(\frac{n}{3}\right) + T\left(\frac{2n}{3}\right) + n.$$

Solution: As in the previous example, we will form levels of the recursive tree of $T(n)$. In the 0 level with $T(n)$ we have n steps that are necessary for a computation of $T(n)$ (if we know $T(\frac{n}{3})$ and $T(\frac{2n}{3})$.

In the first level, $T(n)$ breaks to computations of $T(\frac{n}{3})$ and $T(\frac{2n}{3})$. For this we need $\frac{n}{3} + \frac{2n}{3}$ steps.

In the second level, $T(\frac{n}{3})$ breaks to $T(\frac{n}{9})$ and $T(\frac{2n}{9})$; node $T(\frac{2n}{3})$ breaks to $T(\frac{2n}{9})$ and $T(\frac{4n}{9})$. The sum at the level 2 is then

$$\frac{n}{9} + \frac{2n}{9} + \frac{2n}{9} + \frac{4n}{9} = n.$$

The last non empty level corresponds to $\frac{2^i n}{3^i}$; it ends if and only if $\frac{2^i n}{3^i} = 1$. (As the first one, it ends the branch when $\frac{n}{3^i} = 1$.) Hence the sum in levels in between are smaller that n. The last non empty level corresponds to such i for which

$$n \to \frac{2n}{3} \to \frac{2^2 n}{3^2} \to \ldots \to \frac{2^i n}{3^i} = 1,$$

i.e. $(\frac{2}{3})^i n = 1$, so $n = (\frac{3}{2})^i$ and $i = \log_{\frac{3}{2}} n$.

Therefore

$$T(n) \le n \log_{\frac{3}{2}} n, \text{ hence } T(n) \in \mathcal{O}(n \lg n).$$

1.3.7 Amortized complexity. It is an average of the number of steps of n consecutive operations/instructions in the worst case analysis. If n repetitions in the worst case needs time $\mathcal{O}(T(n))$ then one of them needs time $\mathcal{O}(T(n))/n$, and it is the amortized complexity of one operation/instruction.

There are three methods how to compute the amortized complexity.

- The first one is *Aggregate Method* — we proceed just according the definition.
- The second one is *Accounting Method* — to each operation/instruction a credit is given. If the operation/instruction needs less than the credit, the remaining part of the credit can be used later when more than one credit is needed. The necessary condition is that any time there must be enough credits to cover each operation/instruction.
- The third one is *Potential Method.* Denote by D_i situation after the i-the instruction. So we have a sequence of n situations (usually of data structure) $D_0, \ldots, D_{n-1}$. To every situation D_i a non negative number is associated, so called potential, $\Phi(D_i)$. Denote by c_i the actual price of the transition from D_{i-1} to D_i. Then the amortized price $\hat{c}_i$ corresponding to D_i is defined by
$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}).$$
Then we have
$$\sum_{i=1}^{n} \hat{c}_i = \sum_{i=1}^{n} (c_i + \Phi(D_i) - \Phi(D_{i-1})) = \sum_{i=1}^{n} c_i + \Phi(D_n) - \Phi(D_0).$$
Hence we get conditions on potentials, for every i necessarily $\Phi(D_i) \ge \Phi(D_0)$.

1.3.8 During the lecture, calculation of amortized complexity will be shown using all three methods on the example of the following pseudo code

```
INCREMENT($A$)
1. $i = 0$
2. while $i < A.length$ a $A[i] = 1$
3.         $A[i] := 0$
4.         $i := i + 1$
5. if $i < A.length$
6.         $A[i] := 1$
```

Chapter 2

Time Complexity and Correctness of Algorithms

2.1 Time complexity of algorithms

We show a computation of time complexity on the example of Euclid's algorithm, which fro two positive natural numbers finds their greatest common divisor.

2.1.1 Euclid's Algorithm. Let us calculate the time requirements of the Euclid's algorithms, which for given two natural numbers, finds their greatest common divisor.

The following is a recursive version of Euclid's algorithm:

```
EUCLID($a, b$)
1. if $b = 0$
2.        return $a$
3. else return EUCLID($b, a(\bmod b)$)
```

2.1.2 Time estimation of Euclid's Algorithm. We will use the recursive definition to determine the time requirements.

Lemma 1: If $a > b \geq 1$ and the algorithm Euclid(a, b) needs k recursive calls, then $a \geq F(k+2)$ and $b \geq F(k+1)$, where $F(i)$ is the i-th member of the Fibonacci sequence.

Note that the Fibonacci sequence is defined by:

$$F(0) = 0,\ F(1) = 1,\ F(n) = F(n-1) + F(n-2)\ \text{ for } n \geq 2.$$

Proof (by mathematical induction):

1. Basic step: For $k = 1$, we have $b \geq 1 = F(2)$ and $a > b \geq 1$, i.e. $a \geq 2 = F(3)$.

2. Inductive step: Assume that the assertion holds for $k - 1 \geq 1$ recursive calls. Consider that for numbers $a > b \geq 1$ we need $k \geq 2$ recursive calls. The procedure Euclid(a, b) first calls Euclid$(b, a(\bmod b))$ which needs $k - 1$ recursive calls. From the induction assumption we know that $b \geq F(k+1)$ and $z = a(\bmod b) \geq F(k)$. We have $z = a - qb$ for a suitable integer q and $z < b$. Since $z < b$, we get $q \geq 1$ and therefore

$$a = qb + z \geq qF(k+1) + F(k) \geq F(k+1) + F(k) = F(k+2).$$

Lemma 2: Euclid$(F(k+2), F(k+1))$ needs k recursive calls.

Lemma 3: For every $n \geq 0$ we have $F(n+2) \geq \left(\frac{3}{2}\right)^n$.

Proof. Mathematical induction.

1. Basic step. The assertion is obvious for $n = 0$ and $n = 1$, since $F(2) = 1 \geq \left(\frac{3}{2}\right)^0$ and $F(3) \geq \left(\frac{3}{2}\right)^1$.

2. Inductive step. Assume that $F(n) \geq \left(\frac{3}{2}\right)^{n-2}$ and $F(n+1) \geq \left(\frac{3}{2}\right)^{n-1}$. Then

$$F(n+2) = F(n+1) + F(n) \geq \left(\frac{3}{2}\right)^{n-2} + \left(\frac{3}{2}\right)^{n-1} = \left(\frac{3}{2}\right)^n \left(\frac{2}{3} + \frac{4}{9}\right) \geq \left(\frac{3}{2}\right)^n.$$

To finish the proof it suffices to notice that $\left(\frac{2}{3} + \frac{4}{9}\right) = \frac{10}{9}$.

2.1.3 Proposition: The algorithm $\textsc{Euclid}(a, b)$ requires $\mathcal{O}(\lg b)$ of recursive calls. Hence the complexity related to the number of integer divisions is linear (because the size of its input is proportional to $\lg(a + b)$).

2.1.4 Remark. The upper bound of time complexity of Euclid's algorithm can be proved also by the following consideration.

Proposition. Denote by x_k and y_k the pair of numbers $x_k > y_k$ after k-th recursive call. Then $y_{k+2} < \frac{y_k}{2}$.

Proof. We know that $y_{k+2} < y_{k+1} < y_k$. If $y_{k+1} \leq \frac{y_k}{2}$, then $y_{k+2} < y_{k+1}$ implies that $y_{k+2} < \frac{y_k}{2}$.

Assume that $y_{k+1} > \frac{y_k}{2}$. Then

$$y_{k+2} < x_{k+1} - y_{k+1} = y_k - y_{k+1} < \frac{y_k}{2}.$$

2.2 Correctness of Algorithms

2.2.1 To verify that an algorithm is correct we need to show two things:

1. that the algorithm will halt for every input;
2. when the algorithm halts it gives a correct output – a solution.

Below are a few examples to demonstrate the concepts of the two conditions of 1.4.1.

2.2.2 Bubble Sort.

Input: a sequence of natural numbers $a[1], a[2], \ldots, a[n]$.

Output: the sequence sorted in non-decreasing order.

```
begin
    for $k = n$ step -1 to 2 do
        for $j = 1$ step 1 to $k - 1$ do
            if $a[j] > a[j+1]$ then
                interchange $a[j]$ and $a[j+1]$
end
```

2.2.3 The algorithm 2.2.2 terminates since the outer cycle is executed $(n-1)$-times.

2.2.4 Proposition. After execution of the i-th outer cycle, i.e. for $k = n - i$, we have

a) $a[n-i+1], a[n-i+2], \ldots, a[n]$ are the biggest numbers from numbers $a[1], a[2], \ldots, a[n]$;

b) and $a[n-i+1] \leq a[n-i+2] \leq \ldots \leq a[n]$.

This proposition can be proved using induction on the number of executed cycles.

1. Basic step: For $i = 0$, i.e. before any outer cycle is performed, we have $n - i + 1 = n + 1$ and $a[n+1]$ does not exist. So the assertions hold for it. For $i = 1$, i.e. after one execution of the outer cycle, we have that $a[n-1+1] = a[n]$ and it is the greatest element of the sequence.

2. Induction step: If the assertions hold before the k-th execution of the inner cycle the after it we have $a[n-k+1] \geq a[j]$ for $j \leq n-k$, and a) holds. Moreover, $a[n-k]$ is smaller or equal to $a[n-k+1]$, $a[n-k+2]$, $\ldots$, $a[n]$.

2.2.5 Correctness of the Euclid's Algorithm 2.1.1 Note that even if $a < b$, then after the first division we have $r > t$. Moreover, $t > r \,(\mathrm{mod}\, t) = z$, hence the sequence of ts is strictly decreasing, so it must equal 0 after a finite number of divisions — see condition a).

2.2.6 Proposition. The pair r, t and the pair t, z from Euclid's Algorithm 2.1.1 have the same common divisors.

From the proposition condition b) follows.

2.2.7 Variant. To prove that an algorithm terminates on every input we find a value which is called a *variant.* A variant is usually a natural number that decreases during the run of the algorithm untill it reaches the smallest possible value.

For the Bubble Sort 2.2.2, it is the number k; for Euclid algorithm 2.1.1, it is the remainder z when r is divided by t.

2.2.8 Invariant. An *invariant*, also *conditional correctness of an algorithm*, is an assertion that

- holds before executing the first cycle of the algorithm or after the first execution of the cycle,
- if it holds before an execution of the cycle, then it holds also after its execution,
- when the algorithm terminates, the invariant guarantees that the solution is correct.

The invariant for the Bubble Sort is proposition 2.2.4, for Euclid's Algorithm proposition 2.2.6.

2.3 Minimal spanning trees

2.3.1 A minimal spanning tree in a weighted undirected graph. Given a connected undirected graph $G = (V, E)$ with the set of vertices V and edges E. Let a valuation a of edges be given, i.e. a mapping $a\colon E \to \mathbb{N}$. The aim is to find a spanning tree K of G such that

$$\sum_{e \in K} a(e) \quad \text{is the smallest possible.}$$

We show that any algorithm based on the following scheme is correct.

2.3.2 A general scheme.

Input: a connected undirected graph $G = (V, E)$ and valuation of edges a.

Output: set of edges K of a minimal spanning tree.

```
1. (Initialize)
   $K := \emptyset$, $\mathcal{S} = \{\{v\} \mid v \in V\}$;
2. (Choice of an edge.)
   While $\mathcal{S}$ contains more than one element
          choose an edge $e \in E \setminus K$ such that
                 it joins two distinct sets from $\mathcal{S}$, denote them $C_1$, $C_2$, and
                 for at least one of them it is the cheapest edge leaving it.

3. (Changes.)
   $K := K \cup \{e\}$;
   $\mathcal{S} := (\mathcal{S} \setminus \{C_1, C_2\}) \cup \{C_1 \cup C_2\}$.
```

2.3.3 Termination of the scheme above (a variant). The general scheme above is not an algorithm – the way how we choose an edge e in step 2 is not specified. It we implement it in such a way that it guarantees that it is found in finite amount of time, the the scheme terminates. Indeed, after processing an edge chosen in step 2 the number of elements of $\mathcal{S}$ decreases by one. Since $\mathcal{S}$ has at the beginning n elements, after $n-1$ steps 3 $\mathcal{S}$ has one element and the scheme terminates.

2.3.4 Proposition (an invariant). If the set of edges K before step 2 is a subset of edges of some minimal spanning tree, and if we choose e according to the scheme 2.3.2, then the set $K \cup \{e\}$ is also a subset of edges of some minimal spanning tree.

Proof: Assume that the set K constructed by the scheme 2.3.2 is a subset of a minimal spanning tree T_{min}. Take an edge e from step 2. Then either $e \in T_{min}$ or $e \notin T_{min}$.

The first case is easy: if $e \in T_{min}$, then $K \cup \{e\} \subseteq T_{min}$.

Assume the later possibility; $e \notin T_{min}$. We know that $e = \{u, v\}$ joins two components of connectivity of the graph (V, K), we denote them by S_1 and S_2; say $u \in S_1$ and $v \in S_2$. Further, assume that e is the cheapest edge leaving S_1. Since minimal spanning tree with edges T_{min} is a connected graph, there is a path C in T_{min} from u to v. Denote by e_1 the edge of C which leaves S_1.

Since e is the cheapest edge leaving S_1 and e_1 leaves S_1 as well, we have $a(e) \leq a(e_1)$.

On the other hand, if we add one edge to a spanning tree we close just one circuit; i.e. $T_{min} \cup \{e\}$ contains a circuit which is $C \cup \{e\}$. Hence $T = (T_{min} \cup \{e\}) \setminus \{e_1\}$ forms a spanning tree as well. Moreover, $a(T) = a(T_{min}) + a(e) - a(e_1)$. Since T_{min} is a minimal spanning tree, we have

$$a(T_{min}) + a(e) - a(e_1) \geq a(T_{min}), \text{ tj. } a(e) \geq a(e_1).$$

Thus $a(e) = a(e_1)$ and hence $a(T) = a(T_{min})$. This means that T is also a minimal spanning tree and $K \cup \{e\} \subseteq T$.

2.3.5 Remark. Kruskal's algorithm, and Prim's algorithm are special cases of the general scheme 2.3.2.

2.4 Shortest paths

2.4.1 Shortest paths in a weighted directed graph Let $G = (V, E)$ be a simple, directed weighted graph and $a\colon E \to \mathbb{Z}$ be a mapping on the edges of G to the integers. Denote by n the number of vertices of G, i.e. $n = |V|$.

We will call the number $a(e)$, $e \in E$, the length of the edge e. Let C be a directed walk in G. The *length* of the walk C is $\sum_{e \in C} a(e)$, i.e. the sum of lengths of all edges belonging to C.

2.4.2 Matrix A of lengths is a square matrix $\mathbf{A} = (a(i,j))$ of order n, where n is the number of vertices of G, and

$$a(i,j) = \begin{cases} 0, & \text{for } i = j \\ a(e), & \text{for } e = (i,j) \in E \\ \infty, & \text{for } (i,j) \notin E \end{cases}$$

2.4.3 Matrix U of distances is a square matrix $\mathbf{U} = (u(i,j))$ of order n, where n is the number of vertices of G, and

$$u(i,j) = \begin{cases} 0, \quad \text{for } i = j, \\ \text{the length of a shortest path from } i \text{ to } j, \text{ if there is a path from } i \text{ to } j \\ \infty, \quad \text{if there is no path from } i \text{ to } j \end{cases}$$

2.4.4 Observation. Assume that there is a directed walk from vertex x to vertex y in graph G, then:

1. If G contains only positive length cycles and there exists a shortest walk from x to y, then it is also the shortest path from x to y.
2. If G contains no cycles of negative length, then any shortest walk from x to y has the same length as a shortest path from x to y.
3. If G contains no cycles of negative length, then for every walk C from x to y there exists a path from x to y, that is not longer than the walk C.

2.4.5 Triangular Inequality. If a graph G contains no cycles of negative length, then for every three vertices x, y, z, it holds that

$$u(x,y) \leq u(x,z) + u(z,y).$$

Proof: If $u(x,y) = \infty$ or $u(z,y) = \infty$ then the above inequality holds.

Let C_1 be a shortest path from x to z and C_2 a shortest path from z to y. If we join these two paths together we obtain a walk from x to y whose length is $u(x,z) + u(z,y)$. Since G contains no cycles of negative length, this walk contains a path which is shorter or of the same length. Therefore the length of a shortest path from x to y must satisfy $u(x,y) \leq u(x,z) + u(z,y)$.

2.4.6 Bellman' Principle of Optimality. If a graph G contains no cycles of negative length, then for every three vertices x, y, z, we have

$$u(x,y) = \min_{z \neq y}(u(x,z) + a(z,y)).$$

Proof: Clearly, the above assertion holds for all x, y for which there is no path from x to y.

Assume that there is a path from x to y, i.e. $u(x,y) < \infty$. Since $u(z,y) \leq a(z,y)$ for every two vertices z, y, we know from the Triangle Inequality that $u(x,y) \leq u(x,z) + a(z,y)$. Hence

$$u(x,y) \leq \min_{z \neq y}(u(x,z) + a(z,y)).$$

Choose z to be the second to last vertex in a shortest path from x to y. It is easy to see that for such z the equality holds.

2.4.7 Shortest paths from a given vertex r. Problem: Find lengths of shortest paths from r to every vertex of G.

2.4.8 General Scheme.

Input: a simple directed graph $G = (V, E)$ with lengths a.

Output: values $U(v)$ that equal to $u(r, v)$.

```
1. (Initialization.)
   $U(r) := 0$, $U(v) := \infty$ for $v \neq r$;
2. (Processing of an edge.)
   If there is an edge $e = (v, w)$ such that
         $U(w) > U(v) + a(e)$
               then put $U(w) := U(v) + a(e)$.
3. (Termination.)
   If $U(w) \leq U(v) + a(e)$ for every edge $e = (v, w)$, halt.
   Else go to step 2.
```

2.4.9 Proposition. If a graph G contains no cycles of negative length and a value $U(v) \neq \infty$, then $U(v)$ is the length of some path from r to v.

Outline of the proof: Denote by $U_t(y)$ the value $U(y)$ at time t. We have: if for some time t_k, $U_{t_k}(x) < \infty$, then there is a walk

$$r = v_1, e_1, v_2, e_2, \ldots, v_{k-1}, e_{k-1}, v_k = x$$

and times $t_1 < t_2 < \ldots < t_k$ such that

$$U_{t_i}(v_i) = \sum_{j=1}^{i} a(e_j).$$

Now, it is necessary to prove that the walk is in fact a path, i.e. that it does not contain any vertex twice. Assume the contrary: if for for $i < j$ we have $v_i = v_j$, then $U_{t_i}(v_i) > U_{t_j}(v_j)$. Moreover, $U_{t_j}(v_j) = U_{t_i}(v_i) + a(e_i) + \ldots + a(e_{j-1})$ and therefore $\sum_{k=i}^{j-1} a(e_k) < 0$ and the part of $v_i, e_i, v_{i+1}, e_{i+1}, \ldots, v_j$ contains a cycle of negative length — a contradiction.

2.4.10 Theorem. If graph G contains no cycles of negative length and the value $U(v)$ was calculated according to 2.4.8 then $U(v) = u(r, v)$.

Proof: We proceed by contradiction. Assume that the assertion of the theorem does not hold; i.e. General Scheme 2.4.8 halted and there is a vertex v such that $U(v) > u(r, v)$; this can happen only for $u(r, v) < \infty$.

Consider a shortest path C from r to v. The first vertex of this path is r and it satisfies $U(r) = u(r, r)$; the last vertex of the path is v for which $U(v) > u(r, v)$. Take the first edge $e = (x, y)$ on the path C for which $U(x) = u(r, x)$ and $U(y) > u(r, y)$ (such an edge must exist and is unique). We have:

$$U(y) > u(r, y) = u(r, x) + a(x, y) = U(x) + a(x, y).$$

This is a contradiction with the fact that the general scheme 2.4.8 halted; the Triangle Inequality does not hold for the edge $e = (x, y)$.

2.4.11 Algorithms. We introduce two algorithms that follow the general scheme. Algorithm I systematically tests whether for every edge $e = (v, w)$ it holds that $U(w) \leq U(v) + a(e)$ and halts it is so.

Algorithm I.

Input: A directed graph $G = (V, E)$ with weights a and a vertex r.

Output: $U(v)$ that equal to $u(r, v)$.

```
1. (Initialization.)
   $U(r) := 0$, $U(v) := \infty$ for $v \neq r$
2. (Processing edges.)
   For every edge $e \in E$ do
           if $U(KV(e)) > U(PV(e)) + a(e)$
                   then $U(KV(e)) := U(PV(e)) + a(e)$
3. (Termination.)
   If during the work of 2 no value $U(v)$ was changed, halt and return $U(v)$
   Else go to 2.
```

Algorithm II maintains a set M of vertices x such that it is not evident whether for every edge $e = (x, w)$ the inequality $U(w) \leq U(x) + a(e)$ holds. After initialization, $M = \{r\}$, and the algorithm halts when $M = \emptyset$.

Algorithm II.

Input: A directed graph $G = (V, E)$ with weights a and a vertex r.

Output: $U(v)$ that equal to $u(r, v)$.

```
1. (Initialization.)
   $U(r) := 0$, $U(v) := \infty$ for $v \neq r$; $M := \{r\}$
2. (Processing edges..)
   While $M \neq \emptyset$, choose $x \in M$;
           $M := M \setminus \{x\}$
           for every edge $e$ with $PV(e) = x$ do
                   if $U(KV(e)) > U(x) + a(e)$
                   then $U(KV(e)) := U(x) + a(e)$; $M := M \cup \{KV(e)\}$
3. (Termination.)
   return $U(v)$; halt
```

2.4.12 Shortest paths between all pairs of vertices. Our task is to find the whole matrix $\mathbf{U}$ (instead of only one of its rows).

Denote the set of vertices of G by $V = \{1, 2, \ldots, n\}$. Floyd's Algorithm (also called the Floyd-Warshall Algorithm) constructs matrices $\mathbf{U}_k = (u_k(i, j))$ of order n for $k = 0, 1, \ldots, n$ with the following properties:

$$u_k(i, j) \text{ is the length of a shortest } i - j \text{ path with inner vertices in } 1, 2, \ldots, k.$$

2.4.13 Proposition. It holds that

1. $\mathbf{U}_0$ is the matrix $\mathbf{A}$ of lengths 2.4.2.
2. $\mathbf{U}_n$ is the matrix $\mathbf{U}$ of distances 2.4.3.
3. The matrix $\mathbf{U}_{k+1}$ can be obtained from the matrix $\mathbf{U}_k$ as follows:

$$u_{k+1}(i, j) = \min\{u_k(i, j), u_k(i, k + 1) + u_k(k + 1, j)\}.$$

Proof: The first two properties easily follow from the definition of $\mathbf{U}_0$ and $\mathbf{U}_n$.

The third property follows from the following observation: a shortest path from i to j that goes only through vertices $1, 2, \ldots, k + 1$ either does not contain $k + 1$ (and then has length $u_k(i, j)$) or it goes through $k + 1$ just once (and then is of length $u_k(i, k + 1) + u_k(k + 1, j)$) .

2.4.14 Floyd's Algorithm.

Input: matrix $\mathbf{A}$ of lengths.

Output: matrix $\mathbf{M} = \mathbf{U}$ of distances.

```
1. [Initialization]
        $\mathbf{M} := \mathbf{A}$
2.      begin
             for $k = 1, 2, \ldots, n$ do
                 for $i = 1, 2, \ldots, n$ do
                     for $j = 1, 2, \ldots, n$ do
                         begin
                             if $M(i,j) > M(i,k) + M(k,j)$ then
                                 $M(i,j) = M(i,k) + M(k,j)$
                         end
        end
```

2.4.15 Floyd's Algorithm terminates after a finite number of steps, since the outer cycle is executes n-times, i.e. the variant is k which ranges from 1 to n.

The invariant is 2.4.12 and the property 3 from 2.4.13.

2.5 Huffman Code

2.5.1 Huffman code for lossless data compression. Given data containing characters from an alphabet C; moreover, for every character $c \in C$ its frequency $c.freq$ is given (i.e. how often the character c occurs). We can code characters either by binary words of the same length (so called *fixed-length code*); length of a codeword is given by the number of symbols. The length is then the smallest k such that $|C| \leq 2^k$. In that case, the length of encoded data is the product of the number of characters and the length of a code word k.

Another possibility is to code characters by binary words of distinct length (so called *variable-length code*). Then the length of coded data equals to

$$\sum_{c \in C} c.freq \cdot |w(c)|,$$

where $w(c)$ is the code word of character c and $|w(c)|$ is its length. Note, that this is the same formula also for a fixed-length code.

Encoding is always easy, decoding is easier if no codeword is a prefix of another codeword. Such codes are called *prefix codes* (less often *prefix-less codes*). Any fixed-length codes is a prefix code. We will study only variable-length codes that are prefix codes.

Every prefix binary code we can described as a binary tree T where leaves are labeled by characters from C, edges are labeled by 0 and 1, where 0 means "go to the left" and 1 means "go to the right". A codeword of a character c is the label of the directed path from the root to the leaf c.

The length of coded data is

$$B(T) = \sum_{c \in C} c.freq \cdot d_T(c),$$

where $d_T(c)$ is the depth of the leaf c in the tree T.

Huffman code is a binary variable-length prefix code T with the smallest value of $B(T)$.

2.5.2 Constructing a Huffman code.

Input: Given an alphabet C, $n = |C|$, and frequencies $c.freq$ of characters $c \in C$ in data.
Output: Tree T of an optimal binary tree.

1. Construct n trees T_c each with one element (root) labeled by $c; c.freq$;
   $Q := C$; $\mathcal{T} := \{T_c \mid c \in C\}$.
2. While $|Q| \neq 1$, choose $x \in Q$ with the smallest value $x.freq$, and $y \in Q$ with the second smallest value $y.freq$;
   add anew element z into Q, put $z.freq := x.freq + y.freq$, and x, y remove from Q;
   Construct a tree T_z with root z (labeled $z; z.freq$) as follows: the left subtree of z is T_x, the right subtree of z is T_y;
   remove trees T_x and T_y from $\mathcal{T}$, add T_z to $\mathcal{T}$.
3. For $Q = \{q\}$ and $\mathcal{T} = \{T_q\}$, T_q is a binary tree which determines a prefix binary code by: each edge into a left successor is labeled by 0, each edge to the right successor is labeled by 1. Put $T := T_q$.

2.5.3 Variant. Every execution of step 2 decreases the number of elements of Q (and of $\mathcal{T}$ as well) by one. Therefore, algorithm halts after $n-1$ executions of step 2.

2.5.4 Invariant.

Proposition. Let C be an alphabet and $c.freq$, $c \in C$, frequencies of characters in data. Let x and y are two characters with the smallest frequencies. Let $C' = (C \setminus \{x, y\}) \cup \{z\}$, where $z.freq = x.freq + y.freq$, all other frequencies are the same. Let T' be an optimal tree (i.e. a tree with the smallest value $B(T')$) for C'.

Construct a tree T from T' by replacing the leaf z by a three element tree, its root is z, the left successor of z is x, and the right successor of z is y.

Then T is an optimal tree for C.

Idea of a proof. It can be proved that if we form a new tree T (for alphabet C) from T' (for alphabet C') in such a way that: the leaf z with $z.freq = x.freq + y.freq$ is replaced by three element tree with root z, its left successor x, its right successor y, then

$$B(T) = B(T') + x.freq + y.freq.$$

Now, to finish the proof it suffices to know that it is always possible to find an optimal code for the alphabet C in such a way that the codewords for characters x and y have the same length and they differ only in the last bit. And this is what the following lemma states.

2.5.5 Lemma. Given an alphabet C with frequencies $c.freq$. Let x and y be two characters with the smallest frequencies. Then there exists an optimal variable-length code for which the codewords for x and y have the same length and they differ only in the last bit.

Idea of a proof. Denote by T the tree of an optimal code and denote by a, b the characters of C which are in the last level of T, have the same father, and $a.freq \leq b.freq$. Then $x.freq \leq a.freq$ and $y.freq \leq b.freq$.

If $x.freq = b.freq$ then all characters x, y, a, b have the same frequency and we can exchange x with a ans y with b without changing the value B.

Assume that $x.freq \neq b.freq$. We form a new tree T' by exchanging x with a and y with b. It can be calculated that

$$B(T) - B(T') = (a.freq - x.freq)(d_T(a) - d_T(x)) + (b.freq - y.freq)(d_T(b) - d_T(y)).$$

The expression on the right hand side is non negative. It cannot be positive, otherwise T is not optimal — the tree T' would have smalled value $B(T')$); hence T' is also optimal; which proves the lemma.

Chapter 3

Turing Machines

At first, we study a classical model that preceded modern informatics and due to it the progress of computers. It is so called a Turing machine introduced by Alan Turing in thirties of the twentieth century.

3.1 Deterministic Turing Machines

3.1.1 Informal description of a Turing machine. Informally, can picture if as follows: It consists of

- a *finite control*, which can be in one of a finite number of different states,
- a potentially infinite tape that is divided into cells, with each cell holding one of the tape symbols, and
- from a head that reads the content of a cell and writes a tape symbol in the cell.

According to tape symbol X, which is read by the head, and according to state q of the finite control, the Turing machine changes its state to p, writes a tape symbol Y on the scanned tape cell, and moves its head either left or right. This change is described by the transition function δ.

3.1.2 A formal definition. *A Turing machine*, shortly TM, is a seven-tuple $(Q, \Sigma, \Gamma, \delta, q_0, B, F)$ where

- Q is a finite set of states,
- Σ is a finite set of input symbols (inputs),
- Γ is a finite set of tape symbols where $\Sigma \subset \Gamma$,
- B is a *blank*, a special symbol meaning that the cell is empty; it is a tape symbol that is not an input (i.e. $B \in \Gamma \setminus \Sigma$),
- δ is a transition function, i.e. a partial mapping from $(Q \setminus F) \times \Gamma$ into $Q \times \Gamma \times \{L, R\}$, (where L means that the head moves one cell to the left, R one cell to the right),
- $q_0 \in Q$ is an initial state, and
- $F \subseteq Q$ is a set of final or accepting states.

3.1.3 Instantaneous Description. Instantaneous description (ID), also called a configuration, for a Turing machine M fully describes the state of M, the content of the tape, and the position of its head. Any time during the work of M, the tape contains only a finite number of cells that have a non-blank tape symbol.

Assume that

- expeting k cells with tape symbols $X_1, X_2, \ldots, X_k$, all cells with smaller or greater number contain B,
- the finite control is in state q and the head is scanning symbol X_i.

Then we say that M is in ID

$$X_1\, X_2\, \dots\, X_{i-1}\, q\, X_i\, X_{i+1}\, \dots\, X_k.$$

3.1.4 Initial ID. At the beginning of the work of TM over an input word $w \in \Sigma^\star$, $w = a_1 \dots a_n$, the finite control is in state q_0, the tape contains $a_1 \dots a_n$ in n consecutive cells and the head is scanning the symbol a_1; other cells contain B. It means that the initial ID is

$$q_0\, a_1\, a_2 \dots a_n.$$

3.1.5 A move of a TM. Given a Turing machine $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$.

Assume that $\delta(q, X_i) = (p, Y, R)$, then

$$X_1\, X_2\, \dots\, X_{i-1}\, q\, X_i\, \dots\, X_k \vdash X_1\, X_2\, \dots\, X_{i-1}\, Y\, p\, X_{i+1}\, \dots\, X_k. \tag{3.1}$$

If $i = k$ then

$$X_1\, \dots\, X_{k-1}\, q\, X_k \vdash X_1\, \dots\, X_{k-1}\, Y\, p\, B.$$

Assume that $\delta(q, X_i) = (p, Y, L)$, then

$$X_1\, X_2\, \dots\, X_{i-1}\, q\, X_i\, \dots\, X_k \vdash X_1\, \dots\, X_{i-2}\, p\, X_{i-1}\, Y\, X_{i+1}\, \dots\, X_k. \tag{3.2}$$

If $i = 1$ then

$$q\, X_1\, \dots\, X_k \vdash p\, B\, Y\, \dots\, X_k.$$

If $\delta(q, X_i)$ is not defined then the Turing machine halts.

3.1.6 Computation. A *computation of a TM* over an input word $w = a_1 a_2 \dots a_k$ is a sequence of its moves starting with the initial ID $q_0\, a_1\, \dots\, a_k$.

We will call the last ID, say $\alpha\, p\, \beta$, of a computation over w the *result* of the computation, and we will write

$$q_0\, a_1\, \dots\, a_k \vdash^\star \alpha\, p\, \beta.$$

Note, that $\vdash^\star$ is the transitive and reflexive closure of the relation $\vdash$ from 3.1.5 (on the set of all IDs of the Turing machine).

If for $w \in \Sigma^\star$ the TM enters a state $p \in F$, then we say that TM *halts successfully*, and the content of the tape is called the *output* of TM corresponding to w.

If a TM cannot make a move and the finite control of TM is not in a final state, we say that TM *halts unsuccessfully.*

3.1.7 Definition — a language accepted by a Turing machine. Given a TM $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$. We say that a word $w \in \Sigma^\star$ is *accepted* by M if M successfully halts over w.

Language accepted by M consists of all words accepted by M. In other words,

$$L(M) = \{w \in \Sigma^\star \mid q_0 \vdash^\star \alpha p \beta, \;\; \text{where } p \in F,\ \alpha, \beta \in \Gamma^\star\}.$$

3.1.8 Definition — a function realized by a Turing machine. Given a mapping $f\colon \Sigma^\star \to \Sigma^\star$. We say that M *realizes mapping* f if:

- for every $w \in \Sigma^\star$ for which $f(w)$ is defined M sucessfully halts with output $f(w)$;
- for every $w \in \Sigma^\star$ for which $f(w)$ is not defined M halts unsuccessfully.

Assume that f is a mapping $f\colon \mathbb{N}^k \to \mathbb{N}$, i.e. it assings to an k-touple of natural numbers $n_1, n_2, \dots n_k)$ a natural number $f(n_1, \dots, n_k)$. Then M is . M *realizes* f if:

- if $w = 0^{n_1} 1 0^{n_2} 1 \dots 1 0^{n_k}$ then M halts successfully with the output $0^{f(n_1,\dots,n_k)}$;
- if w is not of the form $0^{n_1} 1 0^{n_2} 1 \dots 1 0^{n_k}$, then M halts unsuccessfully.

3.1.9 Remark. In the above definition 3.1.7 nothing is said about words $w \notin L(M)$; for $w \notin L(M)$, M can either halt unsuccessfully or M may not halt at all. We introduce the notion of *decidable* languages to define when M is guaranteed to halt.

Definition. We say that M *decides* a language L if $L = L(M)$ and M halts on every input word w.

Clearly, if M decides L then M also accepts L. The converse does not hold — later we shall see an example of a language L that is accepted by a TM but there exists no TM that decides L.

3.1.10 Remark. In some literature, even when dealing with Turing machine as acceptor it is required that when a word is accepted, the Turing machine has to have first cell to be scanned. Similarly, when a function is realized then when halting it is required that the head scans the first symbol of $f(w)$. We do not require anything like this. It is easy to see that the difference is only technical — when entering our "accepting state" the new Turing machine moves left till the first blank and then moves right and halts.

There are other variants of Turing machines: some models work with a semi infinite tape — the tape has a fixed left end. In this case, at the start of execution, the input word is written at the beginning of the tape and the head scans the first cell. In this model TM unsuccessfully halts even if the head scans the first cell and the required move is to the left.

Another version of a Turing machine allows its head to stay stationary, i.e. to scan the same cell. In this case the transition function δ is a partial mapping from $(Q \setminus F) \times \Gamma$ to $Q \times \Gamma \times \{R, L, S\}$, where S means that in the next move the head reads the same cell.

All these models are equivalent: whenever a language L is accepted/decided by a Turing machine of one type there is a Turing machine of the other type that accepts/decides L.

3.1.11 Time and space complexity of a Turing machine.

Definition. *Time complexity of M* is a partial function $T(n)$ from the set of all natural numbers such that

- If for a word w of length n the Turing machine M does not halt, $T(n)$ is not defined.
- Otherwise, it is the maximum number of steps that M does before halting, where the maximum is taken over all words $w \in \Sigma^\star$ of length n.

Definition. *Space complexity of M* is a partial function $S(n)$ from the set of all natural numbers such that

- If for a word $w \in \Sigma^\star$ of length n the Turing machine M needs infinitely many tape cells, $S(n)$ is not defined.
- Otherwise, $S(n)$ is the maximum difference between numbers of tape cells visited by M, where the maximum is taken over all words $w \in \Sigma^\star$ of length n.

3.1.12 Techniques for design of TM. There are two basic techniques — information remembered by a state, and multiple tracks. In the first case, the name of a state can contain an information. Indeed, a name of a state can be a pair $(q, 0)$ or $(q, 1)$ with the meaning that $(q, 0)$ is entered if 0 was read, unlike $(q, 1)$ which is entered if 1 was read.

For simplifying a design of a TM we can use the techniques called "multiple tracks". Not only name of a state can carry an information, also tape symbols can be pairs, triples, k-tuples. In such case we say that the tape has 2, 3, or k tracks.

3.2 Turing machines with k tapes.

Informally, a *Turing machine M with k tapes* consists of a finite control, k infinite tapes and k independent heads. According to the state in which the finite control is, and according to the k

tape symbols that the heads read, M changes the state of the finite control, writes k tape symbols on the appropriate tape cells and move k heads independently either to the left or to the right.

Definition. Formally, a TM M with k tapes consists of:

- a finite control which is in one of a finite number of states $q \in Q$,
- a finite number of input symbols Σ,
- a finite set of tape symbols Γ,
- a transition function δ, which is a partial mapping $\delta\colon (Q \setminus F) \times \Gamma^k \to Q \times \Gamma^k \times \{L, R\}^k$,
- an initial state q_0,
- and a set $F \subseteq Q$ of final or accepting states.

At the beginning we have:

- the input $w = a_1 a_2 \dots a_n$ is on the first tape surrounded by blanks, and all other tapes contain only blanks;
- finite control is in q_0;
- the first head scans a_1.

3.2.1 A move of a Turing machine with k tapes. If the transition function is defined then, (according to δ):

- finite control changes the state;
- each head rewrites the content of the cell that is scanned;
- each head independently moves either one cell to the right or to the left.

3.2.2 The language accepted/decided by a Turing machine M with k tapes is defined similarly as for a TM with one tape. We define that M *successfully halts* if the finite control of M enters an accepting state, i.e. a state $p \in F$. If M cannot make the next move, i.e. the transition function is not defined for a given state and given k-tuples of tape symbols, then we say that M halts *unsuccessfully.*

A word $w \in \Sigma^\star$ is *accepted* by TM M if during the work over w M successfully halts. The set of all words accepted by M is called *the language accepted by* M and is denoted by $L(M)$.

Moreover, if M halts on every input word $w \in \Sigma^\star$, then we say that $L(M)$ is *decided* by M.

3.2.3 Remark. Every Turing machine with one tape can be considered as a Turing machine with k tapes for $k = 1$.

3.2.4 Theorem. For every Turing machine M_1 with k tapes there exists a Turing machine with one tape M_2 which accepts/decides the same language as M_1.

Moreover, if M_1 needs for n moves then M_2 needs $\mathcal{O}(n^2)$ moves.

Idea of its proof. The Turing machine M_2 has one tape with $2k$ tracks. Each tape of M_1 is simulated by two tracks of M_2 – in such a way that the first track (corresponding to the tape) contains only the position of the head of M_1, the second track contains the content of the tape which is simulated.

One step of TM M_1 is modeled:

The head of TM M_2 is in such position that all the cells containing information about heads of M_1 is to its right. At first, the head of M_2 scans all the positions of of heads of M_1, the contents read by them are retained in the state. Hence, M_2 has got all the information for one move of M_1.

According to the transition function of TM M_1 when traversing to the left, it changes contents of the even tracks and moves the margins in odd tracks accordingly (if the head of M_1 moves to the left/right, so does the corresponding marker).

If the step we simulate is the n/th step of TM M_1 then M_2 needs for simulating it need at most $\mathcal{O}(n)$ steps.

3.3 Nondeterministic Turing Machines

If a Turing machine (either with one tape or k tapes) is allowed to move from one ID to several different IDs, then we get *nondeterministic* Turing machine, shortly NTM.

Definition. *Nondeterministic Turing machine*, shortly NTM, is a seven-tuple $(Q, \Sigma, \Gamma, \delta, q_0, B, F)$ where

- Q is a finite set of states,
- Σ is a finite set of input symbols,
- Γ is a finite set of tape symbols where $\Sigma \subset \Gamma$,
- B is blank, the empty symbol, $B \in \Gamma \setminus \Sigma$,
- δ is a transition function: a mapping from the set $(Q \setminus F) \times \Gamma$ into the set $\mathcal{P}_f(Q \times \Gamma \times \{L, R\})$ ($\mathcal{P}_f(X)$ is a set of all finite subsets of X),
- $q_0 \in Q$ is a initial state and
- $F \subseteq Q$ is a set of final/accepting states.

A move of an NTM is defined analogously as for a deterministic TM:

Assume that $(p, Y, R) \in \delta(q, X_i)$, then

$$X_1\, X_2\, \ldots\, X_{i-1}\, q\, X_i\, \ldots\, X_k \vdash X_1\, X_2\, \ldots\, X_{i-1}\, Y\, p\, X_{i+1}\, \ldots\, X_k. \tag{3.3}$$

Assume that $(p, Y, L) \in \delta(q, X_i)$, then

$$X_1\, X_2\, \ldots\, X_{i-1}\, q\, X_i\, \ldots\, X_k \vdash X_1\, \ldots\, X_{i-2}\, p\, X_{i-1}\, Y\, X_{i+1}\, \ldots\, X_k. \tag{3.4}$$

3.3.1 The language L that is accepted by an NTM consists of all words $w \in \Sigma^\star$ for which

$$q_0\, w \vdash^\star Y_1\, Y_2\, \ldots Y_i\, q_f\, Y_{i+1}\, \ldots\, Y_m$$

for some accepting state $q_f \in F$ ($\vdash^\star$ is the transitive and reflexive closure of $\vdash$).

Informally: a word w is accepted by a nondeterministic Turing machine if and only if there exists an *accepting computation*, i.e. a sequence of moves after which the finite control enters an accepting state. (Notice, that there may also exist a computation over w which does not halt successfully and/or that does not halt at all.)

If a nondeterministic Turing machine M accepts a language L, and, moreover, if every computation of M over any input halts always after a finite number of moves, then we say that M *decides* the language L.

3.4 Random Access Machine — RAM

3.4.1 In this section we introduce another formal model of an algorithm — the Random Access Machine (RAM), which is closer to a ”classical” computer than a Turing machine. We will show that these two models are equivalent, in other words, that what can be done by a program on RAM can be done by TM and vice versa. Moreover, a simulation of a Turing machine by RAM can be done in such a way that the amount of moves TM needs to simulate n steps of the program on RAM is $\mathcal{O}(n^3)$. This will allow us to switch between these two models whenever it will be useful.

3.4.2 Random Access Machine, or just RAM, consists of a control unit, an arithmetic unit, a memory unit and an input and an output unit.

3.4.3 Program unit contains a program and a program register (the program register points to an instruction which should be performed).

3.4.4 Arithmetic unit executes arithmetic operations such as addition, subtraction, multiplication and division.

3.4.5 Memory is divided into memory cells; each cell can contain an integer. We assume that there is an unlimited number of memory cells and that there is no limit to the size of an integer contained in a cell. The index of a cell is called its *address*.

The cell with address 0 is called the *working register*.

3.4.6 Input unit consists of an input tape and a head. The input tape is divided into cells; in each cell a positive integer can be stored. At any moment the input head is scanning exactly one cell. After a content of a cell is scanned, the head moves one cell to the right.

3.4.7 Output unit consists of an output tape and a head. Similar to the case of input unit, the tape is divided into cells. If the output head writes a number into a cell, it moves one cell to the right.

3.4.8 Configuration of the RAM is a mapping that assigns an integer to every input and output cell, each memory cell and to the program register (only finite number of cells can be nonzero).

Initial configuration is a configuration for which there is a natural number n with the following properties:

- except for the first n input cells, all the other cells contain 0
- the program register contains number 1
- the first n cells contain the input of the RAM

3.4.9 Computation on an input w of RAM is a sequence of configurations such that the first configuration is the initial one and every following configuration was created according to the program of RAM.

3.4.10 Program of a RAM uses the following instructions:

- shift instructions: LOAD operand, STORE operand,
- arithmetic instructions: ADD operand, SUBTRACT operand, MULTIPLY operand, DIVIDE operand
- input and output instructions: READ, WRITE
- jump instructions: JUMP label, JZERO label, JGE label
- halt instructions: HALT, ACCEPT, REJECT

3.4.11 Operand is either a number j, denoted $= j$, or the content of the j-th memory cell, we write j, or the content of the memory cell with address c_j, where c_j is the content of the cell with address j, denoted $*j$.

3.4.12 Label is a natural number which means the number of the instruction to be performed if there is a jump.

3.4.13 Time complexity. We say that a program P for RAM has time complexity $\mathcal{O}(f(n))$ if for every input of size n the number $T(n)$ of steps of RAM is $\mathcal{O}(f(n))$.

3.4.14 Space complexity. We say that a program P for RAM works with space size m if during the computation no instruction with the address of an operand greater than m was performed and at least one instruction was used on an operand with address m.

We say that program P has space complexity $\mathcal{O}(g(n))$ if for every input of size n program P works with space size $\mathcal{O}(g(n))$.

3.4.15 Remark. If there is an input on which the program P does not halt, then the time complexity is undefined. If there is not bound on the address then the space complexity is undefined.

3.4.16 Theorem. For every Turing machine M, there is a program P for RAM such that P simulates the behaviour of M. Moreover, if M needs n moves then P needs $\mathcal{O}(n^2)$ steps.

3.4.17 Theorem. For a program P of a RAM, there exists a Turing machine M with five tapes such that P and M have the same behavior.

3.4.18 Theorem. If a program P of a RAM satisfies the following conditions:

- P contains only instructions that increase the length of a number written in binary by at most 1;
- P contains only instructions that a Turing machine with l tapes can perform with words of length k in $\mathcal{O}(k^2)$ moves,

then the Turing machine from the theorem 3.4.17 simulates n steps of P using $\mathcal{O}(n^3)$ of its moves.

3.4.19 Corollary. Given a program P of a RAM which satisfies conditions from 3.4.17, there is a Turing machine with one tape that has the same behavior as P and n steps of P are simulated by $\mathcal{O}(n^6)$ moves of the Turing machine.

Chapter 4

Complexity Classes

4.1 Decision Problems

The complexity theory deals mainly with so called *decision* problems. These are problems for which a *solution* is either "YES" or "NO".

Example. *SAT – Satisfiability of Boolean formulas:* Given a formula φ in CNF (conjunctive normal form). Decide whether φ is satisfiable.

The answer for a given φ is either "YES" if for φ there is a truth valuation in which it is true, or "NO" if φ is not true in any truth valuation. Note that we do not ask for a truth valuation in which it is true, only its existence is important.

4.1.1 Different types of problems. Lot of problems from real live are of a different type than the example above. Often they are problems the aim of which is to find a feasible solution that is a "best" one. Usually, for each instance I of a problem there is a set of feasible solutions $F(I)$, and a goal function d which to every feasible solution assigns a real number. Three types of questions can be asked:

- Find a feasible solution for which the goal function is optimal. (Whether the word optimal means maximal or minimal depends on the problem) — optimization version.
- Find the value of the goal function of an optimal feasible solution — evaluation version.
- Given moreover a constant K. Decide whether there is a feasible solution with the value of its goal function nor worse than K — decision version.

We show all three types of problems on the example of traveling salesman problem.

4.1.2 Traveling Salesman Problem – TSP. Given towns 1,2, $\ldots, n$. For each pair of towns i, j, a number $d(i, j)$ is given (so called distance between i, j).

Informally, a tour is the order of towns in which a salesman can visit so that he visits each just once and he returns to the town in which he started. Then length of the tour is the sum of all distances that he made during the tour.

Formally: A *tour* is given by a permutation π of $\{1, 2, \ldots, n\}$. *Length of tour* T corresponding to permutation π is

$$d(T) = \sum_{i=1}^{n-1} d(\pi(i), \pi(i+1)) + d(\pi(n), \pi(1)).$$

Optimization version of TSP: Find a tour T for which $d(T)$ is the smallest one.

Evaluation version of TSP: Find the length of an optimal tour T, i.e. a tour with smallest length.

Decision version of TSP: Moreover, a number K is given. Decide whether ther is a tour T for which

$$d(T) \leq K.$$

4.1.3 Decision versions.

- A minimal spanning tree: Given a connected undirected graph $G = (V, E)$, valuation $c: E \to \mathbb{N}$, and a number L. Is there a minimal spanning tree with its length at most L?
- Given a matrix of lengths $\mathbf{A} = (a(i,j))$, two vertices r, c, and a number L. Is there a path from r to c of length at most L?

4.1.4 Evaluation versions.

- A minimal spanning tree: Given a connected undirected graph $G = (V, E)$ and a valuation $c: E \to \mathbb{N}$. What is the length of a minimal spanning tree?
- Given a matrix of lengths $\mathbf{A} = (a(i,j))$ and two vertices r, c. Find the length of a shortest path from r to c.

4.1.5 Optimization versions.

- A minimal spanning tree: Given a connected undirected graph $G = (V, E)$ and a valuation $c: E \to \mathbb{N}$. Find a minimal spanning tree.
- Given a matrix of lengths $\mathbf{A} = (a(i,j))$ and two vertices r, c. Find a shortest path from r to c.

4.1.6 It can be proved that if any of the three versions of an optimization problem is polynomially solvable then so is its optimization version as well. We show such justification on the example of the Traveling salesman problem.

Suppose that there is a polynomial algorithm $\mathcal{A}$ which decides for a given instance of TSP and given number L whether there is a tour of length at most L.

Consider an instance I of TSP. Denote by d the maximum of $d(i,j)$; further denote $A := n \cdot d$ where n is the number of towns. Call $\mathcal{A}$ for $L := \lceil \frac{A}{2} \rceil$. It $\mathcal{A}$ answers "YES" then for a new number L choose the mid point between 0 and L, if $\mathcal{A}$ answers "NO" then new L becomes K the mid point between L and $2L$. We repeat this procedure till the interval for new L becomes trivial (of length 0). Now, the last value of L is the length of an optimal tour and this is the solution of the evaluation version of TSP. Note that since we dealt only with **integer** value of L, the procedure stops after $\lg(A) = \lg(n \cdot d)$ which is $\mathcal{O}(\lg(n))$ repetitions.

We have shown that after $\mathcal{O}(\lg(n))$ calls of the algorithm $\mathcal{A}$ we know the optimal length of a tour, we denote it by D_{opt}.

Consider the complete undirected graph G on the set of vertices $V = \{1, \ldots, n\}$ with edge lengths $d(i,j)$. Now, we give directions to the edges as follows: for $i < j$ the edge $\{i,j\}$ becomes a directed edge (i,j). We order edges lexicographically. Now, for every edge (i,j) in this order we create a new instance $I_{i,j}$ of TSP in such a way that in the previous instance we change the value of $d(i,j)$ to $d(i,j) := n \cdot d$. Further, we call the algorithm $\mathcal{A}$ for $I_{i,j}$ and $L = D_{opt}$. If $\mathcal{A}$ answers "YES" we leave $d(i,j) := n \cdot d$. If $\mathcal{A}$ answers "NO, $d(i,j)$ gets its original value. And we pass to the next pair (i,j) in the ordering.

When we have only n pairs with the original value $d(i,j)$, these n edges form an optimal tour of the instance I.

Notice that in the second part we have used only $\mathcal{O}(n^2)$ calls of the algorithm $\mathcal{A}$. Therefore we get: If there was a polynomial algorithm solving the decision version of TSP, then there is a polynomial algorithm that solves the optimization version of TSP.

4.2 Classes $\mathcal{P}$ and $\mathcal{NP}$

4.2.1 Instance of a problem as a word over a suitable alphabet. Every instance of an arbitrary decision problem can be encoded as a word over a suitable alphabet. We will demonstrate this on the SAT problem and the problem of finding shortest paths in a given weighted graph.

- An instance of SAT (Satisfiability Problem) is an arbitrary formula φ in conjunctive normal form (CNF). Rename logical variables of φ by x_1, x_2, $\ldots, x_n$. Then we can encode φ as a word over the alphabet $\{x, 0, 1, (,), \vee, \wedge, \neg\}$ as follows: x_i will correspond to the word xw where w is the number i written in binary; other symbols remain the same.

  For example, the following formula $\varphi = (x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_4)$ corresponds to the following word
  $$(x1 \vee \neg\, x10 \vee x11) \wedge (\neg\, x1 \vee x100).$$

- Consider the problem of finding a shortest path from a vertex r to a vertex c, an instance consists of a matrix of lengths $\mathbf{A}$, two vertices r and c, and a number k. Let us rename the vertices of the graphs by numbers, i.e. $V = \{1, 2, \ldots, n\}$. $\mathbf{A}$ can be easily coded as a word; this will be followed by the vertices r and c, and a number k (written in binary); the matrix, r, c, and k are separated by a symbol $\#$.

4.2.2 A decision problem as a language over an alphabet. Given a decision problem $\mathcal{U}$, since an output (a solution) of $\mathcal{U}$ is either YES or NO, we divide instances of $\mathcal{U}$ into YES-instances and NO-instances. The *language* corresponding to $\mathcal{U}$, denoted by $L_{\mathcal{U}}$, consists of all words that represent YES-instances of $\mathcal{U}$.

Note that some words over the alphabet Σ do not need to correspond to any instance of $\mathcal{U}$. We consider such words as corresponding to NO-instances. Thus we can assume that the set of all NO-instances forms a complement of the language $L_{\mathcal{U}}$, i.e. it equals to $\Sigma^\star \setminus L_{\mathcal{U}}$.

4.2.3 Class $\mathcal{P}$. We say that a decision problem $\mathcal{U}$ (a language L) belongs to the class $\mathcal{P}$ if there is a deterministic Turing machine which decides $L_{\mathcal{U}}$ (L) and has a polynomial time complexity; i.e. its complexity is $\mathcal{O}(p(n))$ for a polynomial $p(n)$.

4.2.4 Examples of $\mathcal{P}$ problems.

- **Minimal spanning tree.** Given an undirected graph G with weight function c, and a number k, does there exist a spanning tree of G with weight at most k?
- **Shortest paths in an acyclic graph.** Given an acyclic graph with weight function a, vertices r and c, and a number k, does there exist a directed path from r to c of length at most k?
- **Flows in networks.** Given a network with capacity c, source s, sink t, and k representing desired flow, does there exist a admissible flow of value at least k?
- **Minimal cut.** Given a network with capacity c and a number k, does there exist a cut of capacity at most k?

The problems listed above are stated as decision problems. Very often we speak about their optimization versions also as polynomially solvable problems.

4.2.5 Class $\mathcal{NP}$. We say that a decision problem $\mathcal{U}$ (a language L) belongs to the class $\mathcal{NP}$ if there exists a nondeterministic Turing machine which decides $L_{\mathcal{U}}$ (L) in polynomial time.

4.2.6 Remark. In the definition 4.2.3 (class $\mathcal{P}$) we could require instead of a Turing machine a program P for RAM (or an algorithm) which solves $\mathcal{U}$ in polynomial time.

It is not straightforward to construct a nondeterministic Turing machine for a given decision problem, even though we know that it should exist. To verify that a problem, i.e. a language, belongs to the class $\mathcal{NP}$, it is useful to introduce a concept similar to an algorithm, or a program for RAM. A nondeterministic algorithm is the notion we will use.

4.2.7 A nondeterministic algorithm works over an instance I in two steps:

Step 1. It randomly generates a sequence s,

Step 2. A (deterministic) algorithm (a Turing machine, a program for RAM) takes as an input I and s and gives an answer either YES or NO.

We say that a nondeterministic algorithm solves a problem $\mathcal{U}$ if

1. For every YES-instance of $\mathcal{U}$ there exists a sequence s for which the step 2 answers YES.
2. For any NO-instance of $\mathcal{U}$ there exists no sequence s for which the step 2 answers YES.

We say that a nondeterministic algorithm *works in time* $\mathcal{O}(T(n))$ if any execution of both steps 1 and 2 for an instance of size n requires $\mathcal{O}(T(n))$ steps.

4.2.8 Remark. The fact that a nondeterministic algorithm works in polynomial time means that also each step requires polynomial time. Therefore, any sequence s generated by step 1 must be of polynomial length (with respect to the size of the instance).

In the definition 4.2.5 we could require instead of a nondeterministic Turing machine an nondeterministic algorithm which solve $\mathcal{U}$ in polynomial time.

4.2.9 Examples of $\mathcal{NP}$ problems.

- **Cliques.** Given an undirected graph G and a number k is there a clique in G having at least k vertices?
- **Shortest paths in a general graph.** Given a directed weighted graph G with weight function a, two vertices r and v, and a number k, is there a directed path from r to v of length at most k?
- **k-colorability.** Given an undirected graph G without loops, is G k-colorable?
- **Knapsack problem.** Given n objects $1, 2, \ldots, n$ each with value c_i and weight w_i, and two natural numbers A and B which represent total weight and total price respectively, is it possible to choose the objects so that their total weight is at most A and their total price is at least B.

  In other words, does there exist a subset of objects $I \subseteq \{1, 2, \ldots, n\}$ such that

$$\sum_{i \in I} w_i \le A \quad \text{a} \quad \sum_{i \in I} c_i \ge B?$$

4.3 Class $\mathcal{NPC}$

4.3.1 Reductions and polynomial reductions. Given two decision problems $\mathcal{U}$ and $\mathcal{V}$. We say that a problem $\mathcal{U}$ *reduces* to a problem $\mathcal{V}$, if there is an algorithm (a program for RAM, a TM) M that for every instance I of $\mathcal{U}$ constructs an instance I' of $\mathcal{V}$ such that

$$I \text{ is a YES-instance of } \mathcal{U} \text{ iff } I' \text{ is a YES-instance of } \mathcal{V}.$$

The fact that $\mathcal{U}$ reduces to $\mathcal{V}$ is denoted by

$$\mathcal{U} \lhd \mathcal{V}.$$

Moreover, if the algorithm M works in polynomial time, then we say that $\mathcal{U}$ *polynomially* reduces to $\mathcal{V}$, and we denote it

$$\mathcal{U} \lhd_p \mathcal{V}.$$

Roughly speaking, $\mathcal{U} \lhd \mathcal{V}$ means that $\mathcal{U}$ is not more difficult than $\mathcal{V}$.

4.3.2 Proposition. Given three decision problems $\mathcal{U}$, $\mathcal{V}$ and $\mathcal{W}$. If

$$\mathcal{U} \lhd_p \mathcal{V} \text{ and } \mathcal{V} \lhd_p \mathcal{W}, \quad \text{then} \quad \mathcal{U} \lhd_p \mathcal{W}.$$

4.3.3 $\mathcal{NP}$ complete problems. We say that a decision problem $\mathcal{U}$ is $\mathcal{NP}$ *complete* if

1. $\mathcal{U}$ belong to the class $\mathcal{NP}$;
2. every $\mathcal{NP}$ problem polynomially reduces to $\mathcal{U}$.

The class of all $\mathcal{NP}$ complete problems is denoted by $\mathcal{NPC}$.

Roughly speaking, $\mathcal{NP}$ complete problems are the most difficult ones among all $\mathcal{NP}$ problems.

4.3.4 Remark. Similarly, we can speak about the class of $\mathcal{NP}$ complete languages. The only difference is that a language L_1 polynomially reduces to a language L_2 means that there is an algorithm M such that for every word w, the algorithm constructs a word w' in such a way that

$$w \in L_1 \quad \text{iff} \quad w' \in L_2.$$

Note that $\mathcal{U} \lhd_p \mathcal{V}$ if and only if $L_{\mathcal{U}} \lhd_p L_{\mathcal{V}}$.

4.3.5 Proposition. Given two $\mathcal{NP}$ problems $\mathcal{U}$ and $\mathcal{V}$ for which $\mathcal{U} \lhd_p \mathcal{V}$, then

1. if $\mathcal{V}$ is in the class $\mathcal{P}$ then so is $\mathcal{U}$;
2. if $\mathcal{U}$ is an $\mathcal{NP}$ complete problem then so is $\mathcal{V}$.

4.3.6 Proposition. If there is an $\mathcal{NP}$ complete problem that belongs to the class $\mathcal{P}$ (i.e. that is polynomially solvable) then $\mathcal{P} = \mathcal{NP}$ and every $\mathcal{NP}$ problem is polynomially solvable.

4.3.7 $\mathcal{NP}$ hard problems. If we know that any $\mathcal{NP}$ problem polynomially reduces to a problem $\mathcal{U}$ (or we know that there is a $\mathcal{NP}$ complete problem that polynomially reduces to $\mathcal{U}$), then we say that $\mathcal{U}$ is $\mathcal{NP}$ hard. Note that this means that $\mathcal{U}$ is at least as difficult as all $\mathcal{NP}$ complete problems.

4.3.8 Cook's theorem. The problem SAT, satisfiability of formulas in conjunctive normal form, is an $\mathcal{NP}$ complete problem.

Sketch of the proof. It is not difficult to verify that SAT belongs to the class $\mathcal{NP}$. The first phase of a nondeterministic algorithm generates a truth valuation u of logical variables contained in φ. It is easy to see that there is a checking algorithm that verifies if φ is true in u. Moreover, any satisfiable formula has a truth valuation u for which it is true, and there exists no such u for an unsatisfiable formula.

The second part of the proof consists of a description of a computation of a nondeterministic Turing machine over a word by means of a formula (which can be transformed to a formula in CNF). We will show the main ideas of such a description.

Given a nondeterministic Turing machine M with the set of states Q, input alphabet Σ, tape alphabet Γ, transition function δ, initial state q_0 and accepting (final) state q_f, assume that M accepts a word w in $p(n)$ moves.

We introduce the following logical variables:

$h_{i,j}$, $i = 0, 1, \ldots, p(n)$, $j = 1, 2, \ldots, p(n)$;
the fact that $h_{i,j}$ is true means that the head of M at timet i reads j-th tape cell.

s_i^q, $i = 0, 1, \ldots, p(n)$, $q \in Q$;
the fact that s_i^q is true means that at time i M is in state q.

$t^A_{i,j}$, $i = 0, 1, \ldots, p(n)$, $j = 1, 2, \ldots, p(n)$, $A \in \Gamma$;

the fact that $t^A_{i,j}$ is true means that at time i the tape contains in j-th cell symbol A.

Now we need formulas to describe the following facts:

1. At every time i, $i = 0, \ldots, p(n)$, NTM M is in exactly one state.
2. At every time i, $i = 0, \ldots, p(n)$, the head of M reads exactly one cell of the tape.
3. At every time i, $i = 0, \ldots, p(n)$, each tape cell contains exactly one tape symbol.
4. At the beginning of the work of M (i.e. at time 0), M is in q_0; its head reads the first tape cell; the tape contains the first n cells the input word, and other cells contain B.
5. One step of M is determined by the transition function; i.e. the state of M at the next time $i+1$, the content of the cell just read, and the cell which is scanned at the time $i+1$ is given by the transition function.
6. The content of tape cells that are not scanned at time i remains the same in the next time $i+1$.
7. If a word is accepted, then at the end of the work, i.e. at time $p(n)$, M is in q_f.

We show how formulas for 1, 4, 5, 6 and 7 can be created.

Add 1. At time i, the Turing machine M is in at least one state:

$$\bigvee_{q \in Q} s^q_i.$$

At time i, the Turing machine M is not in two different states:

$$\bigwedge_{q \neq q'} (\neg s^q_i \vee \neg s^{q'}_i).$$

Now, the fact that M is at time i in exactly one state is the conjunction of both the above formulas:

$$(\bigvee_{q \in Q} s^q_i) \wedge \bigwedge_{q \neq q'} (\neg s^q_i \vee \neg s^{q'}_i).$$

Add 4. At the beginning (i.e. at time 0) M is in the state q_0, the head scans the first tape cell, and the first n cells of the tape contain the input $a_1 a_2 \ldots a_n$; other tape cells contain B.

$$s^{q_0}_0 \wedge h_{0,1} \wedge t^{a_1}_{0,1} \wedge \ldots \wedge t^{a_n}_{0,n} \wedge t^B_{0,n+1} \wedge \ldots \wedge t^B_{0,p(n)}.$$

Add 5. At time i, M is in the state q, the head scans the j-th tape cell that contains the tape symbol A, and $\delta(q, A)$ is $\{(p_k, C_k, D_k) \mid k = 1, \ldots, m\}$ (where $D_k = 1$ means that the head moves to the right, $D_k = -1$ means that the head moves to the left). Then the formula which describes the configuration of M at the next time $i+1$ is:

$$\bigwedge_j \bigwedge_{A \in \Gamma} ((s^q_i \wedge h_{i,j} \wedge t^A_{i,j}) \Rightarrow \bigvee (s^p_{i+1} \wedge t^C_{i+1,j} \wedge h_{i+1,j+D})).$$

Add 6. The content of cells other than the j-th at time $i+1$ remains the same as in i:

$$\bigwedge_j \bigwedge_{A \in \Gamma} ((\neg h_{i,j} \wedge t^A_{i,j}) \Rightarrow t^A_{i+1,j}).$$

Add 7. At the end of the work of M, i.e. at time $p(n)$, M is in the state q_f.

$$s^{q_f}_{p(n)}.$$

The resulting formula is a conjunction of the formulas mentioned above for all times $i = 0, 1, \ldots, p(n)$.

4.4 Reductions

4.4.1 We know, see the proposition 4.3.5, that to prove that a decision problem $\mathcal{U}$ in $\mathcal{NP}$ is $\mathcal{NP}$ complete, it suffices to show that any $\mathcal{NP}$ complete problem polynomially reduces to $\mathcal{U}$. Until now, the only problem which we have proved to be $\mathcal{NP}$ complete is SAT, satisfiability of Boolean formulas in conjunctive normal form. In this section, we show several polynomial reductions that will help us to prove that other decision problems are also $\mathcal{NP}$ complete.

4.4.2 $3 - CNF\ SAT$**.** Problem: Given a formula φ in CNF such that each clause contains at most 3 literals, is φ satisfiable?

4.4.3 **Proposition.** It holds that

$$SAT \lhd_p 3 - CNF\, SAT.$$

Sketch of the reduction SAT **to** $3 - CNF\ SAT$**.** Given a formula φ in CNF, we shall construct a formula ψ such that

1. ψ is in CNF and each clause contains at most 3 literals;
2. ψ is satisfiable if and only if φ is satisfiable.

Denote by $C_1, C_2, \ldots, C_k$ all clauses of φ. if each contains at most 3 literals then $\psi = \varphi$.

For every clause C which contains more than 3 literals we construct a formula ψ_C as follows: Let $C = l_1 \vee l_2 \vee \ldots \vee l_s$, where l_i are literals. We introduce new logical variables $x_1, x_2, \ldots, x_{s-3}$ and put

$$\psi_C = (l_1 \vee l_2 \vee x_1) \wedge (\neg x_1 \vee l_3 \vee x_2) \wedge (\neg x_2 \vee l_4 \vee x_3) \wedge \ldots \wedge (\neg x_{s-3} \vee l_{s-1} \vee l_s).$$

We have: ψ_C is satisfiable iff C is satisfiable.

Formula ψ is the conjunction of all clauses of φ that have at most 3 literals and of formulas ψ_C for clauses C with more than 3 literals.

Assume that φ has k clauses each of them with at most s literals. Then when constructing ψ we created at most $(s-3)k$ new logical variables. Moreover, the number of new literals added to the formula is at most $2(s-3)k$ (each new logical variable appears in ψ twice). Hence the length of ψ is only a polynomial of the length of the original formula φ.

4.4.4 **Corollary.** $3 - CNF\ SAT$ is an $\mathcal{NP}$ complete problem.

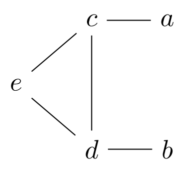
4.4.5 **k-colorability.** Problem: Given a simple undirected graph G without loops and a number k, is G k-colorable?

4.4.6 **Proposition.** It holds that

$$3 - CNF\ SAT \lhd_p \text{3-colorability}.$$

Sketch of the reduction $3 - CNF\ SAT$ **to 3-colorability.** Assume that φ is a formula in CNF where each clause has 3 literals. We have to construct a simple undirected graph G without loops such that φ is satisfiable if and only if G is 3-colorable.

The construction of G uses an auxiliary graph with five vertices $1, 2, 3, 4, 5$ and five edges



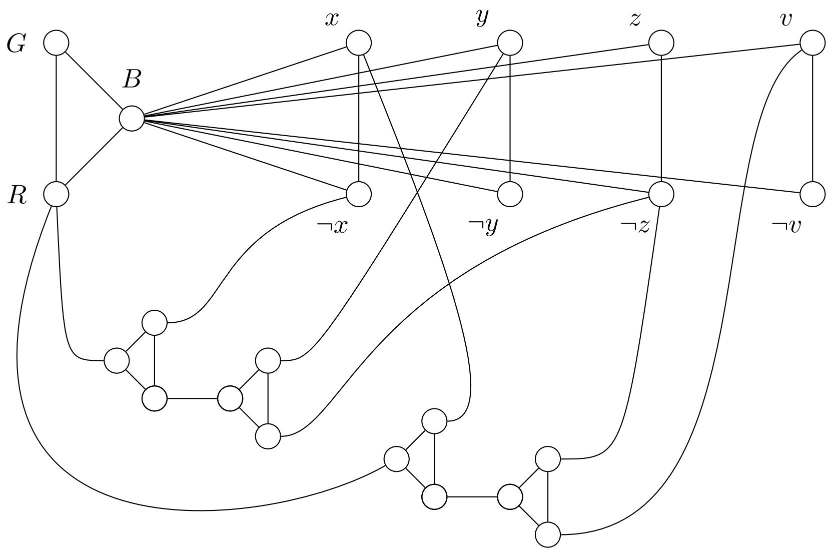
with the following properties:

- If each of vertices 1 and 2 have the same color c, then so must vertex 5.
- If one of vertices 1 and 2 has color g then we can color the graph in such a way that 5 has color g.

Given a formula φ, denote by x_1, x_2, …, x_n all logical variables that occur in φ. Let us construct an undirected graph $G = (V, E)$ where

- V consists of all literals, i.e. $x_1, \neg x_1, \ldots, x_n, \neg x_n$, and vertices R, G, B.
- E contains edges so that vertices R, G, B, $B, x_i, \neg x_i$ for every $i = 1, \ldots, n$ form a triangle.
- Further, for every clause containing literals l_1, l_2, l_3 add to the graph two copies of the auxiliary graph G_1 in the following way: Literals l_2 and l_3 correspond to vertices a and b of the first copy of G_1, vertices l_1 and e correspond to vertices a, b and vertex R is e of the second copy of G_1,

An example of the graph G for two clauses $C_1 = \neg z \vee y \vee \neg x$ and $C_2 = t \vee \neg z \vee x$ $(x, y, z, t$ logical variables) is on the following picture.



Assume that formula φ is satisfiable; then there is a truth valuation in which φ is true. Color the graph G with three colors g (green),r (red) and b (blue) as follows:

- Vertices R, G, B: $c(R) = r$, $c(G) = g$, $c(B) = b$.
- The vertex corresponding to a literal l has color g if and only if l is true, otherwise it has color r.

Since every clause contains at least one literal which is true, i.e. which has color g, it is possible to color all remaining vertices so that G is colored with three colors.

Assume that the graph G is 3-colorable. Rename the colors so that $c(R) = r$, $c(G) = g$, $c(B) = b$. Now, define a truth valuation of logical variables x_1, x_2, …, x_n as follows:

$$\text{variable } x_i \text{ is true iff } c(x_i) = g \text{ and variable } x_i \text{ is false iff } c(x_i) = r.$$

From the properties of the auxiliary graph G_1, it follows that for each clause there is at least one literal with its color g, hence it is true.

It is not difficult to see that the number of vertices and the number of edges of G is polynomial with respect to the length of φ.

4.4.7 Corollary. Since the problem of 3-colorability is in $\mathcal{NP}$, 3-colorability is an $\mathcal{NP}$ complete problem.

4.4.8 Proposition. It holds that

$$\text{3-colorability} \ \lhd_p \ ILP.$$

Reduction of 3-colorability to ILP. Given a simple undirected graph $G = (V, E)$ without loops, we will construct an instance I of integer linear program such that I has a feasible solution if and only if G is 3-colorable. (In fact, variables will have value either 0 or 1, so it will be 0-1 linear program.)

Variables: For each vertex $v \in V$ we introduce three variables:

$$x_v^r, x_v^g, x_v^b.$$

Meaning: The fact that a variable x_v^c equals 1, $c \in \{r, g, b\}$ means that vertex v has color c.

Constraints:

- For every vertex $v \in V$ we have one equation that guarantees that v has exactly one color — either r or g or b:
$$x_v^r + x_v^g + x_v^b = 1.$$
- For every edge $e = \{u, v\}$ we have three inequalities (each for one color). They guarantee that two adjacent vertices u and v do not have the same color:
$$x_u^r + x_v^r \le 1, \ \ x_u^g + x_v^g \le 1, \ \ x_u^b + x_v^b \le 1.$$

We have that G is 3-colorable if and only if I has a feasible solution.

The number of variables of I is $3\,|V|$; further, I has $|V| + 3\,|E|$ constraints. Hence the size of I is $\mathcal{O}(n + m)$ where $n = |V|$ and $m = |E|$.

4.4.9 Corollary. Since ILP belongs to $\mathcal{NP}$, ILP is an $\mathcal{NP}$ complete problem.

4.4.10 Partition problem. Problem: Given a finite (nonempty) set X and a collection of its subsets $\mathcal{S}$.

Question: Is it possible to choose from $\mathcal{S}$ a collection $\mathcal{A}$ that forms a partition of X? In other words, is there $\mathcal{A} \subseteq \mathcal{S}$ such that each element $x \in X$ belongs to exactly one set of $\mathcal{A}$?

4.4.11 Proposition. It holds that

$$\text{3-colorability} \ \lhd_p \ \text{Partition problem.}$$

Reduction of 3-colorability to Partition problem. Given a simple undirected graph $G = (V, E)$ without loops, we will construct a set X and a collection of its subsets $\mathcal{S}$ such that G 3-colorable if and only if $\mathcal{S}$ contains a partition of X.

Set X:

- For every vertex $v \in V$ the set X contains elements
$$v, p_v^r, p_v^g, p_v^b.$$
- For each edge $e = \{u, v\}$ the set X contains elements
$$q_{uv}^r, q_{uv}^g, q_{uv}^b, q_{vu}^r, q_{vu}^g, q_{vu}^b.$$

Hence, X has $4\,|V| + 6\,|E|$ elements.

The system $\mathcal{S}$:

1. For every vertex $v \in V$ the system $\mathcal{S}$ contains:
$$\{v, p_v^r\}, \{v, p_v^g\}, \{v, p_v^b\}.$$
2. Given vertex $v \in V$, denote the neighborhood of v as $N(v)$ (i.e. $N(v) = \{u \,|\, \{v,u\} \in E)$. $\mathcal{S}$ further contains:
$$S_v^r = \{p_v^r, q_{vu}^r \,|\, u \in N(v)\}, S_v^g = \{p_v^g, q_{vu}^g \,|\, u \in N(v)\}, S_v^b = \{p_v^b, q_{vu}^b \,|\, u \in N(v)\}.$$
3. For every edge $e = \{u, v\}$ the system $\mathcal{S}$ contains:
$$\{q_{uv}^r, q_{vu}^g\}, \{q_{uv}^r, q_{vu}^b\}, \{q_{uv}^g, q_{vu}^r\}, \{q_{uv}^g, q_{vu}^b\}, \{q_{uv}^b, q_{vu}^r\}, \{q_{uv}^b, q_{vu}^g\}.$$

$\mathcal{S}$ has $3\,|V|$ sets from 1), $3\,|V|$ sets from 2) and $6\,|E|$ sets from 3).

Assume that G is 3-colorable. Then there is a coloring of V by colors $\{r, g, b\}$. Denote the color of $v \in V$ as $c(v)$. Choose $\mathcal{A}$ from $\mathcal{S}$ as follows:

$\mathcal{A}$ consists of:

1. $\{v, p_v^{c(v)}\}$ for all $v \in V$,
2. $S_v^{c_1}$ a $S_v^{c_2}$ where c_1 and c_2 are the colors different from $c(v)$,
3. $\{q_{uv}^{c(u)}, q_{vu}^{c(v)}\}$ for every edge $e = \{u, v\}$.

Assume that there is a partition $\mathcal{A} \subseteq \mathcal{S}$ of X. Then we define a coloring of vertices of G as follows:
$$c(v) := c, c \in \{r, g, b\}, \quad \text{if and only if} \quad \{v, p_v^c\} \in \mathcal{A}.$$
It is not difficult to show that c is a coloring of V by colors r, g, b.

4.4.12 Corollary. Since the Partition problem belongs to $\mathcal{NP}$, it is an $\mathcal{NP}$ complete problem.

4.4.13 SubsetSum. Problem: Given positive integers a_1, a_2, $\dots, a_n$ and a number K, is it possible to choose a subset of numbers a_1, a_2, $\dots, a_n$ in such a way that their sum is K?

In other words, is there a subset $J \subseteq \{1, 2, \dots, n\}$ such that
$$\sum_{i \in I} a_i = K.$$

4.4.14 Proposition. It holds that
$$\text{Partition problem} \lhd_p \text{SubsetSum}.$$

Reduction of Partition problem to SubsetSum. Given a finite nonempty set X and a collection of its subsets $\mathcal{S}$, rename the elements of X so that $X = \{0, 1, \dots, n-1\}$, and let $\mathcal{S} = \{S_1, S_2, \dots, S_r\}$.

Choose a natural number p greater than r (the number of sets in $\mathcal{S}$). To each $S_i \in \mathcal{S}$ assign a positive number a_i as follows: Denote by χ_{S_i} the characteristic function of $S_i \in \mathcal{S}$; i.e. $\chi_{S_i}(j) = 1$ iff $j \in S_i$. Then
$$S_i \longrightarrow \sum_{j=0}^{n-1} \chi_{S_i}(j)\, p^j = a_i.$$
Finally, put $K = \sum_{i=0}^{n-1} p^i$.

Since $p > r$, it is not difficult to show that
$$\sum_{i \in J} a_i = K \text{ iff } \mathcal{A} = \{S_i \,|\, i \in J\} \text{ is a partition of } X.$$

4.4.15 Corollary. Since SubsetSum belong to $\mathcal{NP}$, it is an $\mathcal{NP}$ complete problem.

4.4.16 Remark. It is not difficult to construct a polynomial reduction of SubsetSum to Knapsack problem. Therefore Knapsack problem is an $\mathcal{NP}$ complete.

4.4.17 Clique problem. Problem: Given a simple undirected graph $G = (V, E)$ without loops and a number k, is there a clique in G with at least k vertices?

4.4.18 Proposition. It holds that

$$3 - CNF\ SAT \lhd_p \text{Clique problem.}$$

Sketch of a reduction of $3 - CNF\,SAT$ to Clique problem. Given a formula φ in CNF that has k clauses $C_1, C_2, \ldots, C_k$, where each clause contains 3 literals, construct a k-partite undirected graph $G = (V, E)$ as follows:

The set of vertices of G consists of k three element sets $V_1, \ldots, V_k$, where V_i contains the three literals from clause C_i. For each literal p of V_i and p' of V_j, $i \neq j$ there is an edge $\{p, p'\}$ in G if and only if p and p' is not a pair of complementary literals (i.e. one is not a negation of the other).

We have that φ is satisfiable if and only if there is a clique in G containing k vertices. (Notice that k is the number of clauses of φ.)

Assume that there is a truth valuation u in which φ is true. Choose in each clause C_i one literal p_i which is true in u. Now the set A containing from each V_i the vertex denoted by p_i, $i = 1, \ldots, k$, is a clique in G with k elements.

Assume that there is a clique A of G with k vertices. Then A contains one vertex from each set V_i, $i = 1, \ldots, k$. Set all of the literals from A to be true (i.e. define a truth valuation u in which all literals from A are true); the truth value of other variables are chosen arbitrarily. Then φ is true in u.

The graph G that was constructed above has its number of vertices equal to its number of literals in φ, i.e. n vertices where n is the length of φ. Since a simple graph with n vertices has $\mathcal{O}(n^2)$ edges, G has a polynomial size with respect to the size of φ. Therefore, it is a polynomial reduction.

4.4.19 Corollary. Since the Clique problem belongs to $\mathcal{NP}$, it is an $\mathcal{NP}$ complete problem.

4.4.20 Independent sets. Given a simple undirected graph $G = (V, E)$ without loops, a set of vertices $N \subseteq V$ is called an *independent set* of G if no edge of G has has each of its vertices in N. In other words, the subgraph induced by N is a discrete graph.
Problem: Given a simple undirected graph G without loops and a number k.
Question: Is there an independent set in G with k vertices?

4.4.21 Proposition. It holds

$$\text{Clique problem} \lhd_p \text{Independent sets.}$$

Reduction of Clique problem to Independent sets. Given a simple undirected graph $G = (V, E)$ without loops, define its opposite graph $G^{op} = (V, E^{op})$ by:

$$\{u, v\} \in E^{op} \text{ if and only if } u \neq v \text{ and } \{u, v\} \notin E.$$

We have: A set $A \subseteq V$ is a clique in G if and only if it is a maximal independent set in G^{op}.

It is a polynomial reduction because a complete graph on n vertices has $\frac{n(n-1)}{2}$ edges. Hence the number of edges in G plus the number of edges of G^{op} is $\mathcal{O}(n^2)$.

4.4.22 Corollary. Since the problem of independent sets belongs to $\mathcal{NP}$, it is an $\mathcal{NP}$ complete problem.

4.4.23 Vertex cover. Given a simple undirected graph $G = (V, E)$ without loops. A subset of vertices $B \subseteq V$ is called a *vertex cover* of G if every edge of G has at least one end vertex in B.

Note that the set of all vertices V is a vertex cover but not very interesting one. Also, if B is a vertex cover, so is any B', $B \subseteq B'$.

Problem: Given a simple undirected graph G without loops, and an integer k.
Question: Does in the graph G exist a vertex cover with k vertices?

4.4.24 Proposition. It holds that

$$\text{independent sets} \lhd_p \text{vertex cover.}$$

4.4.25 A reduction of independent sets to vertex cover. We have: If a set N is an independent set of G then the set $V \setminus N$ is a vertex cover of G. Conversely, if B is a vertex cover of G then the set $V \setminus B$ is an independent set in G.

Therefore, if a simple undirected graph G without loops and an integer k are given, then G has an independent set with k vertices if and only if there is a vertex cover in G with $n - k$ vertices ($n = |V|$ is the number of vertices of G).

4.4.26 Corollary. Since the vertex cover problem is in the class $\mathcal{NP}$, it is an $\mathcal{NP}$ complete problem.

4.4.27 Existence of a Hamiltonian cycle. Given a directed graph G.

Question: Does G have a Hamiltonian cycle? (In other words, is there a cycle containing all vertices of G?)

4.4.28 Proposition. It holds that

$$\text{vertex cover } \lhd_p \text{ existence of Hamiltonian cycle.}$$

4.4.29 Sketch of a reduction of vertex cover to existence of Hamiltonian cycle. On of reductions is based on a special graph H with 4 vertices and 6 directed edges. Graph H has the following property: If H is a part of a Hamiltonian cycle then there are just two way how to visit vertices of H; either all vertices are visited consecutively, or H is visited twice, at first two vertices and later another two vertices.

Assume that a simple undirected graph $G = (V, E)$ without loops and an integer k are given. A directed graph G' can be constructed such that in G there is a vertex cover with k vertices if and only if in G' there is a Hamiltonian cycle.

Informally, the graph G' is constructed as follows: For each edge e of G the graph G' contains one copy of H. Moreover, G' contains also vertices $1, 2, \ldots, k$. So, the number of vertices of G' equals to $4\,|E| + k$. Edges of G' are edges of all copies of H, edges joining copies of H, and edges from and to vertices $1, 2, \ldots, k$. (All together in G' there are at most eight times the number of edges of G plus twice the number of k times the number of vertices of G. So the size of G' is a polynomial with respect to the size of G.

4.4.30 Corollary. Since the existence of a Hamiltonian cycle is in $\mathcal{NP}$, it is an $\mathcal{NP}$ complete problem.

4.4.31 Existence of a Hamiltonian circuit. Similarly as for existence of a Hamiltonian cycle one can prove that the problem of vertex cover polynomially reduces to existence of Hamiltonian circuit. Only the auxiliary graph is more complicated. Hence it can be proved that:

4.4.32 Corollary. Since the existence of a Hamiltonian circuit is in $\mathcal{NP}$, it is an $\mathcal{NP}$ complete problem.

4.4.33 Proposition. It holds that

$$\text{existence of a Hamiltonican circuit } \lhd_p \text{ TSP}.$$

There is a very easy polynomial reduction of the fact above, and it is left for the reader to construct it.

4.4.34 Corollary. Since TSP is in $\mathcal{NP}$, it is an $\mathcal{NP}$ complete problem.

4.4.35 Existence of an open directed Hamiltonian path. It is easy to construct a polynomial reduction from the problem of existence of a Hamiltonian cycle to existence of an open Hamiltonian path. Such reduction will be one of the problems of tutorials. So, also existence of an open Hamiltonian path is $\mathcal{NP}$ complete problem.

4.4.36 Proposition. It holds that

$$\text{Existence of a directed Hamiltonian path } \lhd_p \text{ longest paths in a directed graph.}$$

Also this polynomial reduction is easy and it is left for the reader

4.4.37 Corollary. Since the problem of longest paths in a weighted directed graph is in $\mathcal{NP}$, it is an $\mathcal{NP}$ complete problem.

4.4.38 Proposition. It holds that

$$\text{longest paths in a directed weighted graph } \lhd_p \text{ shortest paths in a directed weighted graph.}$$

Again, a polynomial reduction is left to the reader.

4.4.39 Corollary. Since the problem of shortest paths in a directed weighted graph belong to the class $\mathcal{NP}$, it is an $\mathcal{NP}$ complete problem.

4.5 Heuristics and Approximation Algorithms

4.5.1 Heuristics. If we have to solve a problem which is $\mathcal{NP}$ complete it is not possible to find a correct or optimal solution in a reasonable time; till now we can, within a polynomial time, obtain "reasonably correct" answer or "reasonably good" solution. For this purpose heuristic algorithms that work in a polynomial time are used. Polynomial algorithms for which it can be proved "how far from an optimal solutions" their results are, are also called approximation algorithms.

4.5.2 A heuristics for vertex cover problem — 1. Consider the following heuristic algorithm which for a given undirected graph finds its vertex cover. The algorithm is based on so called greedy strategy.

Input: an undirected graph $G = (V, E)$.

Output: a vertex cover C of G.

```
begin
    $C := \emptyset$
        while $E \neq \emptyset$ do
            choose a vertex $v$ with the biggest degree
            $C := C \cup \{v\}$
            delete $v$ together with all edges incident with it
end
return $C$
```

Even though the algorithm above "looks reasonable", it in some cases finds a vertex cover which has considerably more vertices than an optimal one. What is meant by "considerably more": there is a graph which has a vertex cover with k vertices but the algorithm above finds a vertex cover with $\Theta(k \lg k)$ vertices.

4.5.3 A heuristics for vertex cover problem — 2. There is another heuristic algorithm for the vertex cover problem.

Input: an undirected graph $G = (V, E)$.

Output: a vertex cover C of G.

```
begin
      $C := \emptyset$
            while $E \neq \emptyset$ do
                choose an edge $\{u, v\}$
                $C := C \cup \{u, v\}$
                delete vertices $u, v$ together with all edges incident with it
end
return $C$
```

It is not difficult to show that the vertex cover produced by the algorithm above has at most twice the number of vertices than an optimal vertex cover of G.

4.5.4 Proposition. Denote by C_{min} a vertex cover with the minimum number of vertices of G. The the second heuristics find a vertex cover C such that

$$|C| \leq 2\,|C_{min}|.$$

Proof. Denote by F the set of all edges that were chosen by the algorithm during its work. Then $|C| = 2\,|F|$; indeed, for every edge chosen, we put two its vertices into C — the end vertices of this edge. Moreover, no two edges of F share a common vertex; therefore to cover edges of F we need $|F|$ vertices. Hence, $|C_{min}| \geq |F|$ and $|C| = 2\,|F| \leq 2\,|C_{min}|$.

4.5.5 Approximation algorithms. Definition. Consider an optimization problem $\mathcal{U}$. A polynomial algorithm $\mathcal{A}$ is called and R *approximation algorithm* if the number R satisfies: For each instance i of the problem $\mathcal{U}$ the algorithm $\mathcal{A}$ returns a feasible solution of I whose value is not worse than R times the value of an optimal solution.

"Not worse than" means that for a minimization version the value of it is at most R times bigger than the value of an optimal solution, for maximization problem the value is at most R times smaller than the value of an optimal solution.

In fact, the second heuristics is 2 approximation algorithm of the problem of vertex cover, since evidently it is a polynomial algorithm.

Not for all problems the decision version of them are NP complete approximation algorithms exist (if we assume that $\mathcal{P} \neq \mathcal{NP}$). TSP can serve as one of the examples.

4.5.6 Propositon. If there is a number r and a polynomial algorithm $\mathcal{A}$ such that for every instance I of TSP $\mathcal{A}$ returns a tour of length $D \leq r\,OPT(I)$ (where $OPT(I)$ is the length of an optimal tour for I) then

$$\mathcal{P} = \mathcal{NP}.$$

Proof. We show that if an algorithm from the proposition 4.5.6 exists then we would be able to solve the problem of existence of a Hamiltonian circuit in polynomial time. Let us sketch the arguments

Given an undirected graph $G = (V, E)$, $V = \{1, 2, \ldots, n\}$; our task is to decide whether G contains a Hamiltonian circuit. Let us construct an instance of TSP as follows: For towns $\{1, 2, \ldots, n\}$ define

$$d(i,j) = \begin{cases} 1, & \{i,j\} \in E \\ r\,n+1, & \{i,j\} \notin E \end{cases}$$

Any tour of the instance above has length n if and only if it contains only edges of length, i.e. edges of the graph G and the tour is in fact a Hamiltonian circuit. If a tour contains at least one edge which does not belong to G then its length is at least $n - 1 + n\,r + 1 = n\,r + n$.

Hence, if the algorithm $\mathcal{A}$ returns a tour different than n, then in G there is no Hamiltonian circuit. Indeed, $r\,n + n$ is greater than $r\,n$. We have shown that we would be able to polynomially decide whether G has a Hamiltonian circuit or not. Since the problem of existence of a Hamiltonian circuit is $\mathcal{NP}$ complete, it would mean that $\mathcal{P} = \mathcal{NP}$.

4.5.7 Remark. If an instance I of a TSP satisfies the triangle inequality, i.e. if it holds that for every towns i, j, k we have:

$$d(i,j) \le d(i,k) + d(k,j),$$

then there is a 2 approximation and even $\frac{3}{2}$ algorithm for TSP.

4.6 Class co-$\mathcal{NP}$

4.6.1 Observation. If a language L belongs to $\mathcal{P}$, then so does its complement $\overline{L}$. It is not known if an analogous assertion also holds for languages that belong to $\mathcal{NP}$.

4.6.2 Definition. A language L belongs to the class co-$\mathcal{NP}$ if its complement is in $\mathcal{NP}$.

4.6.3 Examples.

- The language $USAT$, which is a complement of the language SAT (of satisfiable Boolean formulas), belongs to $co - \mathcal{NP}$. (The language $USAT$ consists of all unsatisfiable Boolean formulas and all words that do not correspond to any Boolean formula.)

- The language $TAUT$ consists of all words representing tautologies of the propositional logic and belongs to $co - \mathcal{NP}$.

4.6.4 Question whether co-$\mathcal{NP} = \mathcal{NP}$ is an open problem.

4.6.5 Lemma. Given two languages L_1 and L_2 for which $L_1 \lhd_p L_2$. Then also $\overline{L_1} \lhd_p \overline{L_2}$, (where $\overline{L}$ is the complement of the language L).

Proof. The fact that $L_1 \lhd_p L_2$, $L_1 \subseteq \Sigma^\star$, $L_2 \subseteq \Phi^\star$ means that there exists a polynomial algorithm M which for every word $w \in \Sigma^\star$ constructs a word $M(w) \in \Phi^\star$ in such a way that

$$w \in L_1 \quad \text{if and only if} \quad M(w) \in L_2.$$

This means that

$$w \notin L_1 \quad \text{if and only if} \quad M(w) \notin L_2,$$

and thus $\overline{L_1} \lhd_p \overline{L_2}$.

4.6.6 Proposition. It holds that co-$\mathcal{NP} = \mathcal{NP}$ if and only if there is an $\mathcal{NP}$ complete language the complement of which belongs to $\mathcal{NP}$.

Proof. If co-$\mathcal{NP} = \mathcal{NP}$ then every complement of a $\mathcal{NP}$ complete language belongs to $\mathcal{NP}$, therefore there is an $\mathcal{NP}$ complete language the complement of which belongs to $\mathcal{NP}$.

Assume that there is an $\mathcal{NP}$ complete language L whose complement $\overline{L}$ belongs to $\mathcal{NP}$. We show that
co-$\mathcal{NP} \subseteq \mathcal{NP}$ a $\mathcal{NP} \subseteq$ co-$\mathcal{NP}$.

Consider an arbitrary language L_1 from the class co-$\mathcal{NP}$. Then $\overline{L_1} \in \mathcal{NP}$. Since L is $\mathcal{NP}$ complete language, $\overline{L_1} \lhd_p L$, and by the lemma above $L_1 \lhd_p \overline{L} \in \mathcal{NP}$. Threfore $L_1 \in \mathcal{NP}$.

Consider an arbitrary language L_2 for which $L_2 \in \mathcal{NP}$. Then $L_2 \lhd_p L$, and hence (by the lemma above) $\overline{L_2} \lhd_p \overline{L} \in \mathcal{NP}$. Therefore $L_2 \in$ co-$\mathcal{NP}$.

4.7 Classes $\mathcal{PSPACE}$ and $\mathcal{NPSPACE}$

4.7.1 Given a Turing machine M (deterministic or nondeterministic), we say that M works with space complexity $p(n)$, if for every word of length n, M uses at most $p(n)$ tape cells.

4.7.2 Class $\mathcal{PSPACE}$. A language L belongs to $\mathcal{PSPACE}$ if there exists a deterministic Turing machine M that accepts L and works with polynomial space complexity.

4.7.3 Proposition. It holds
$$\mathcal{P} \subseteq \mathcal{PSPACE}.$$

4.7.4 Class $\mathcal{NPSPACE}$. A language L belongs to the class $\mathcal{NPSPACE}$ if there exists a nondeterministic Turing machine M that accepts L and works with polynomial space complexity.

4.7.5 Proposition. It holds
$$\mathcal{NP} \subseteq \mathcal{NPSPACE}.$$

4.7.6 Theorem. If a Turing machine M (deterministic or nondeterministic) accepts a language L with space complexity $p(n)$, then there is a constant $c \geq 1$ such that M accepts any word $w \in L$ of length n after at most $c^{p(n)+1}$ steps.

Idea of the proof. The constant c is chosen so that for an input word of length n the Turing machine M has less than $c^{p(n)+1}$ different IDs. Indeed, if a word w is accepted by M, then it is so with computations that do not enter the same ID twice.

Denote by t the number of tape symbols of M and s the number of states of M. Then M has $p(n)\, s\, t^{p(n)}$ different IDs.

Put $c = t + s$. From the Binomial Theorem we have
$$c^{p(n)+1} = (t+s)^{p(n)+1} = t^{p(n)+1} + (p(n)+1)\, t^{p(n)}\, s + \dots .$$
Therefore, $c^{p(n)+1} \geq p(n)\, t^{p(n)}\, s$.

4.7.7 Theorem. If a language L belongs to $\mathcal{PSPACE}$ ($\mathcal{NPSPACE}$), then L is decided by a deterministic (nondeterministic) Turing machine M with polynomial space complexity that halts always after at most $c^{q(n)}$ steps, where $q(n)$ is a polynomial and c is a constant.

Idea of the proof. Assume that a language L belongs to $\mathcal{PSPACE}$. Then there is a Turing machine M_1 that accepts the language L with space complexity $p(n)$ ($p(n)$ is a suitable polynomial). We know (see 4.7.6), then there is a constant c such that the Turing machine M_1 needs at most $c^{p(n)+1}$ steps.

Construct a Turing machine M_2 that has two tapes: the first one simulates M_1; the second one counts steps made in the first tape. If the number of steps exceeds $c^{p(n)+1}$, then M_2 halts unsuccessfully.

The required Turing machine M is the Turing machine with one tape that simulates M_2. M works with time complexity $\mathcal{O}(c^{2p(n)})$, so it will halt in at most $d\,c^{2p(n)}$ steps. Now, if we put $q(n) = 2p(n) + \log_c d$ or any polynomial bounded below by $q(n)$ (for all n) we get the required statement.

4.7.8 Savitch's Theorem. It holds

$$\mathcal{PSPACE} = \mathcal{NPSPACE}.$$

Idea of the proof. Clearly, $\mathcal{PSPACE} \subseteq \mathcal{NPSPACE}$. Proof of the opposite inclusion $\mathcal{NPSPACE} \subseteq \mathcal{PSPACE}$ is based on the fact that for a nondeterministic Turing machine M we are able to construct a deterministic Turing machine M_1 that decides the language $L(M)$ and works with polynomial space complexity (even though time complexity may be exponential).

Given a nondeterministic Turing machine M that decides L with polynomial space complexity $p(n)$, a deterministic Turing machine that decides the same language as M with polynomial space complexity can be constructed by introducing a Boolean recursive procedure called $REACH(I, J, m)$, where I and J are IDs and m is a number.

$REACH(I, J; m)$

Input: ID's I and J of an NTM M, and m a positive integer.

Output: TRUE if J is reachable from I in at most m steps, otherwise FALSE.

```
begin
   if $m = 1$ then
         if $I = J$ or $I \vdash J$ then return TRUE
         else return FALSE
   end
   else (*inductive part*]
         for every possible ID $K$ do
         if $REACH(I, K; \lfloor \frac{m}{2} \rfloor)$ and $REACH(K, J; \lceil \frac{m}{2} \rceil$ then
             return TRUE
         return FALSE
   end
end
```

Let w be an input word; we call procedure $REACH(I_0, J, m)$ for the initial ID I_0, an accepting ID J of M and $m = c^{p(n)+1}$ (where c is the constant from 4.7.6). It can be proved that a deterministic Turing machine requires only $\mathcal{O}([p(n)]^2)$ tape cells to make all recursive calls of $REACH(I, J, m)$. (Note that we only claim that the space complexity is polynomial; the time complexity may be, and often is, exponential.)

Notice that during the work of the procedure $REACH(I, J; m)$, on the stack there is only at most one triple $(I_1, J_1; m)$, at most one triple $(I_2, J_2; \frac{m}{2})$, at most one triple $(I_3, J_3; \frac{m}{4})$, etc. Thus at the same time, the stack contains at most $\lg m$ distinct triple at one time.

Given a nondeterministic Turing machine M that accepts L with a polynomial space complexity $p(n)$. Given an input word w, we call procedure $REACH(I_0, J; m)$ where I_0 is an initial ID of M, J is an accepting ID of M, and $m = c^{p(n)+1}$ (c is the constant from 4.7.6). It can be proved that for an execution of $REACH(I, J; m)$ by a deterministic Turing machine, space complexity $\mathcal{O}([p(n)]^2)$ is sufficient. Indeed, it follows from the fact that $REACH(I_0, J; m)$ has on it stack at most $\lg c^{p(n)+1} = d\,p(n)$ triples $(I, J; r)$ and each triple has length at most $\mathcal{O}(p(n))$. (Note that we are interested only in space complexity; the deterministic Turing machine can use exponential number of steps.)

4.7.9 Corollary. It holds
$$\mathcal{P} \subseteq \mathcal{NP} \subseteq \mathcal{PSPACE}.$$

4.7.10 Remark. If we define a class $\mathcal{EXP}$ for all languages that are solved by a deterministic Turing machine in exponential number of steps (with respect to the size of an input) then theorem 4.7.7 together with Savitch's Theorem imply that
$$\mathcal{PSPACE} \subseteq \mathcal{EXP}.$$

4.8 Primality Testing

4.8.1 Languages L_p and L_s. Let language L_p be the set of all prime numbers and L_s be the set of all composite numbers; more precisely:
$$L_p = \{w \mid w \text{ is a prime number in binary}\}$$
$$L_s = \{w \mid w \text{ is a composite number in binary}\}.$$
The language L_s is "nearly" the complement of L_p, only the number 1 does not belong to either set. Hence, we add 1 into L_s and get
$$L_s = \overline{L_p}, \quad L_p = \overline{L_s}.$$

4.8.2 Proposition. L_s belongs to $\mathcal{NP}$.

Proof: A number n is composite if and only if there exist integers r and s such that $n = rs$, $r < n$ and $s < n$. If we know r, we can easily verify that r divides n and that $r < n$; the existence of the given r proves that n is composite. Of course an r satisfying the necessary conditions $n = rs$ and $1 < r < n$ does not exist for prime numbers.

The above consideration shows that r is the required certificate of polynomial length. Indeed, the length of any number smaller than n is $\mathcal{O}(k)$, where $k = \lg n$ is the size of the input n. Moreover, integer division of two numbers in binary can be carried out in polynomial time with respect to the length of the binary words, i.e. k.

4.8.3 Corollary. L_p belongs to co-$\mathcal{NP}$.

4.8.4 Proposition. L_p belongs to $\mathcal{NP}$.

Finding a polynomial certificate for the language of all prime numbers is considerably more difficult than for the language of composite numbers. Here is a generator of the group $(\mathbb{Z}_p \setminus \{0\}, \odot, 1)$ (p a prime number); in another words, a primitive element of the finite field $(\mathbb{Z}_p, \oplus, \odot, 0, 1)$.

4.8.5 Corollary. Languages L_p and L_s belong to the intersection of $\mathcal{NP}$ and co-$\mathcal{NP}$.

4.8.6 Furthermore, we show that there is a probabilistic algorithm, called the Miller-Rabin primality test, that for a large odd integer $n$has at least a 50 % chance of deciding if n is prime. Before discussing the test, let us review several facts that will be needed:

- Set $\mathbb{Z}_n$ of classes modulo n consists of
$$\mathbb{Z}_n = \{0, 1, \ldots, n-1\}.$$
- There are two operations $\oplus$ and $\odot$ on $\mathbb{Z}_n$ defined by
$$a \oplus b = c, \text{ where } c \text{ is the remainder when } a + b \text{ is divided by } n,$$
$$a \odot b = c, \text{ where } c \text{ is the remainder when } ab \text{ is divided by } n.$$

- $(\mathbb{Z}_n, \oplus, 0)$ is a commutative group; $(\mathbb{Z}_n, \odot, 1)$ is a commutative monoid. Distributive laws hold between $\odot$ and $\oplus$.

  Moreover, there is an inverse element for $a \in \mathbb{Z}_n$ (with respect to $\odot$) if and only if a and n are relatively prime.

  Hence, $(\mathbb{Z}_n, \oplus, \odot, 0, 1)$ is a field whenever n is prime n. If n is composite, then $(\mathbb{Z}_n, \oplus, \odot, 0, 1)$ is not a field (it has zero divisors).

- Theorem (Small Fermat Theorem): If a and p are relatively prime for prime number p then
$$a^{p-1} \equiv 1 \pmod{n}.$$

- Given a finite group G and its subgroup H, then the number of elements of H divides the number of elements of G.

- Operations addition, multiplication, division and taking powers in $\mathbb{Z}_n$ can be carried out in polynomial time with respect to the size of numbers that are involved in the operations. (Recall that the size of a number n is $\lg n$.)

4.8.7 Miller-Rabin primality test.

Input: a large odd natural number n.

Output: *prime* or *composite*.

```
1. Compute $n - 1 = 2^l m$ where $m$ is odd.

2. Randomly choose $a \in \{1, 2, \ldots, n-1\}$.

3. Compute $a^m \pmod{n}$,
       if $a^m \equiv 1 \pmod{n}$ halt; output prime.

4. By repeated squaring compute
       $a^{2m} \pmod{n}, a^{2^2 m} \pmod{n}, \ldots, a^{2^l m} \pmod{n}$.

5. If $a^{2^l m} \not\equiv 1 \pmod{n}$ halt; output composite.

6. Take $k$ such that $a^{2^k m} \not\equiv 1 \pmod{n}$ and $a^{2^{k+1} m} \equiv 1 \pmod{n}$, then
   if $a^{2^k m} \equiv -1 \pmod{n}$ halt, output prime;
   if $a^{2^k m} \not\equiv -1 \pmod{n}$ halt, output composite.
```

4.8.8 Theorem.

1. If Miller-Rabin primality test for n outputs *composite*, then n is composite.
2. If Miller-Rabin primality test for n outputs *prime*, then n is prime with probability at least $\frac{1}{2}$.

Informal argument for the theorem 4.8.8. Add 1. If n is prime, then Miller-Rabin primality test cannot halt in step 5. Indeed any $a \in \{1, \ldots, n-1\}$ is relatively prime to n and hence, by the Small Fermat Theorem, we have $a^{2^l m} \equiv 1 \pmod{n}$.

Furthermore, if n prime, then $(\mathbb{Z}_n, \oplus, \odot)$ is a finite field and there are only two elements b with $b^2 = 1$, namely $b = 1$ or $b = -1$. Hence the algorithm cannot halt with output *composite*.

Add 2. Showing the other assertion is more complicated. The proof is not difficult for those composite numbers n for which there exists at least one $a \in \mathbb{Z}_n$: a and n relatively prime, and $a^{n-1} \not\equiv 1 \pmod{n}$. For other composite numbers, called *Carmichael numbers*, (also *pseudo primes*), the proof is rather difficult.

We show the main idea of the proof: The proof is based on the fact that the number of a's chosen in step 2 for which the algorithm gives the correct output *composite* is at least as big as the number of a's for which the algorithm gives the incorrect output *prime*. Since each a is chosen with the same probability, it suffices to show that there are more a's for which the output is *composite* than those a's for which the output is *prime*.

Assume that the a chosen in step 2 is not relatively prime to n. Then we must get the output *composite*, since no power of a equals 1 in $\mathbb{Z}_n$.

Assume that a composite number n is not Carmichael, in other words there exists $a \in \mathbb{Z}_n$, a and n relatively prime, and $a^{n-1} \not\equiv 1 \pmod{n}$. Denote this by

$$\mathbb{Z}_n^\star = \{a \in \mathbb{Z}_n \mid a \text{ is invertible}\}$$

$$K = \{a \in \mathbb{Z}_n^\star \mid a^{n-1} = 1\}.$$

We know that $K \neq \mathbb{Z}_n^\star$; at the same time $(K, \odot)$ is a subgroup of the group $(\mathbb{Z}_n^\star, \odot)$. Therefore, the number of elements of K divides the number of elements of $\mathbb{Z}_n^\star$. Hence the number of elements of K is at most half of the number of elements $\mathbb{Z}_n^\star$; in other words

$$|\mathbb{Z}_n^\star \setminus K| \geq |K|.$$

If we choose $a \in \mathbb{Z}_n^\star \setminus K$ we get the correct output *composite*, since $a^{n-1} \neq 1$.

The incorrect output *composite* might be obtained (not necessarily) only for those a's that belong to K, and we have $|K| \leq |\mathbb{Z}_n^\star \setminus K|$.

If n is Carmichael, then $|K| = |\mathbb{Z}_n^\star|$, and we must show that in step 6 there are at least the same number of a's for which we get a square root of 1 different from -1 as the number of a's for which we get -1 as a square root of 1.

4.9 Classes based on randomization

4.9.1 Randomized Turing machines. A RTM is, roughly speaking, a Turing machine M with two tapes in which the first tape has the same role as it does for a deterministic Turing machine and the second tape contains a sequence of 0 and 1 generated equiprobably (contains 0 or 1 with equal probability $\frac{1}{2}$).

At the beginning of its work:

- M is in its initial state q_0;
- the first tape contains the input word w, other cells contain blank B;
- the second tape contains a randomly generated sequence of 0s and 1s;
- other tapes (if any) contain only Bs;
- every head scans the first cell of its tape.

According to the state q in which M is, and to the content of the scanned cells of tapes, the transition function δ of M determines ifr M halts, or M does in one step the following actions:

- changes its state,
- rewrites the content of the first tape (**but does not change the content of second tape**),
- moves each of its heads either to the right or to the left or does not move (the moves of heads are independent).

Formally, $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$, where Q, Σ, Γ, q_0, B and F have the same meaning as they did for a DTM. The transition function δ is a partial function from $Q \times \Gamma \times \{0,1\}^\star$ to $Q \times \Gamma \times \{L, R, S\}^2$.

If M is in a state q, then the first head scans symbol X, the second head scans number a, and if

$$\delta(q, X, a) = (p, Y, D_1, D_2), \quad q, p \in Q, a \in \{0,1\}, X, Y \in \Gamma, D_1, D_2 \in \{L, R, S\},$$

then M changes its state to p; on the first tape writes Y; and the i-th head moves to the right if $D_i = R$, or to the left if $D_i = L$, or does not move if $D_i = S$.

If $\delta(q, X, a)$ is not defined, then M halts.

M halts successfully if and only if it enters a final (accepting) state $q \in F$.

4.9.2 Remark. The difference between a deterministic Turing machine and a randomized Turing machine is the role of the second tape. A DTM with two tapes can rewrite the contents of both tapes which is forbidden for a RTM. Moreover, if an RTM runs twice on the same input, the two computations may be different (the computation depends on the content of the random tape). This is not possible for a DTM with 2 tapes.

To realize a random tape — we are not able to fill an infinite random tape before starting the work of an RTM — we use the following: whenever a new cell of a random tape is scanned an RTM randomly chooses either 0 or 1 (both with the same probability $\frac{1}{2}$). This symbol never changes during the computation of RTM.

4.9.3 Example. Given a RTM M where $Q = \{q_0, q_1, q_2, q_3, q_f\}$, $\Gamma = \{0, 1, B\}$ and the transition function δ is defined by the following table:

	$0,0$	$1,0$	$0,1$	$1,1$	$B,0$	$B,1$
$\rightarrow q_0$	$(q_1, 0, R, S)$	$(q_2, 1, R, S)$	$(q_3, 0, S, R)$	$(q_3, 1, S, R)$	$-$	$-$
q_1	$(q_1, 0, R, S)$	$-$	$-$	$-$	(q_4, B, S, S)	$-$
q_2	$-$	$(q_2, 1, R, S)$	$-$	$-$	(q_4, B, S, S)	$-$
q_3	$(q_3, 0, R, R)$	$-$	$-$	$(q_3, 1, R, R)$	(q_4, B, S, S)	(q_4, B, S, S)
$\leftarrow q_4$	$-$	$-$	$-$	$-$	$-$	$-$

Assume that the input of M is the word w, then:

- If the first random bit is 0 (i.e. 0 was randomly generated at first), then M checks whether either $w = 0^n$ or $w = 1^n$ for some $n > 0$.
- If the first random bit is 1 (i.e. 1 was randomly generated at first), then the head of the random tape moves to the right and M whether the content of the random tape starting at second cell coincide with the input w.

Otherwise, M halts unsuccessfully.

For an RTM one has to calculate the probability that for a given input w the M halts successfully, i.e. it halts in an "accepting" state q_f. In our example, the answer is the following:

- If w is the empty word then M never halts in q_f (i.e. for no content of the random tape).
- If either $w = 0^n$ or $w = 1^n$ for $n > 0$, then M halts in q_f with probability

$$\frac{1}{2} + \frac{1}{2}\left(\frac{1}{2}\right)^n = \frac{1}{2} + 2^{-(n+1)}.$$

- If w contains both 0 and 1, then the probability that M halts in q_f is

$$\frac{1}{2}\left(\frac{1}{2}\right)^{|w|} = 2^{-(|w|+1)}.$$

4.9.4 Class $\mathcal{RP}$. A language L belongs to $\mathcal{RP}$ if and only if there exists a RTM M such that:

1. If $w \notin L$ then M accepts in state $q_f \in F$ with probability 0.
2. If $w \in L$ then M accepts in state $q_f \in F$ with probability at least $\frac{1}{2}$.
3. There exists a polynomial $p(n)$ such that every computation of M (i.e. for any content of the random tape) M needs at most $p(n)$ steps, where n is the length of an input word.

The Miller-Rabin primality test is an example of an algorithm satisfying the three conditions of 4.9.4 (if we construct a corresponding RTM) and therefore the language L_s which consists of all composite numbers belongs to $\mathcal{RP}$.

4.9.5 Turing machine of type Monte-Carlo. A RTM satisfying conditions 1 and 2 from 4.9.4 is called a Turing machine of type *Monte-Carlo*.

Note that a Monte-Carlo TM does not need to work in a polynomial time.

4.9.6 Proposition. Given a language $L \in \mathcal{RP}$, for every positive constant $0 < c < \frac{1}{2}$ there exists a RTM M (an algorithm) with polynomial complexity such that:

1. If $w \notin L$, then M accepts in state $q_f \in F$ with probability 0.
2. If $w \in L$, then M accepts in state $q_f \in F$ with probability at least $1 - c$.

4.9.7 Class $\mathcal{ZPP}$. A language L belongs to $\mathcal{ZPP}$ if and only if there exists a RTM M such that:

1. If $w \notin L$ then M halts in an accepting state $q_f \in F$ with probability 0.
2. If $w \in L$ then M halts in an accepting state $q_f \in F$ with probability 1.
3. The expected number of steps of M during one computation over an input of length n is $p(n)$, where $p(n)$ is a suitable polynomial.

Roughly speaking: M does not make a mistake, but it is not guaranteed that the number of steps is always polynomial; only the expected number of steps is polynomial.

4.9.8 Turing machine of type Las-Vegas. A RTM satisfying all three conditions from definition 4.9.7 is called a *Las Vegas Turing machine.*

4.9.9 Proposition. If a language L belongs to $\mathcal{ZPP}$, then so does its complement $\overline{L}$.

The same RTM M of type Las-Vegas serves for "accepting" L and well as its complement $\overline{L}$; it suffices to declare accepting states as non-accepting and non-accepting states as accepting ones.

4.9.10 Remark. It is not known if a similar assertion hold for languages in $\mathcal{RP}$.

4.9.11 Class co-$\mathcal{RP}$. A language L belongs to co-$\mathcal{RP}$ if and only if its complement $\overline{L}$ belongs to $\mathcal{RP}$.

4.9.12 Theorem.

$$\mathcal{ZPP} = \mathcal{RP} \cap \text{ co-}\mathcal{RP}.$$

Proof. First, we show that $\mathcal{RP} \cap$ co-$\mathcal{RP} \subseteq \mathcal{ZPP}$.

Assume that a language L belongs to each of classes $\mathcal{RP}$ and co-$\mathcal{RP}$. Then there exist two RTM's M_1 and M_2 each of type Monte-Carlo with polynomial time requirements such that

M_1 — is for L;

M_2 — is for $\overline{L}$.

Denote by $p(n)$ the polynomial that is an upper bound for the number of steps for M_1 and M_2. Construct a RTM M for the language L as follows: Given an input word w,

1. M executes M_1 for $p(n)$ steps. If M_1 accepts, then M accepts (enters its accepting state).
2. M executes M_2 for $p(n)$ steps. If M_2 accepts, then M rejects (enters a non accepting state and halts).
3. If M does not halt in step 1 or 2, M repeats 1 and 2. (until it halts either in 1 or 2.).

It can be proved that the above M does not make a mistake and, moreover, it is of type Las-Vegas.

Now let us prove that $\mathcal{ZPP} \subseteq \mathcal{RP} \cap$ co-$\mathcal{RP}$.

Assume that L belongs to $\mathcal{ZPP}$. Thus there is a RTM M_1 of type Las-Vegas for L. Denote by $p(n)$ the polynomial that bounds the expected number of steps of M_1 for any input of length n. We construct a RTM M of type Monte-Carlo with polynomial time requirements for L as follows:

Given an input w; M executes M_1 for $2p(n)$ steps. If M_1 accepts, then so does M; in all other cases M halts without accepting.

It can be proven that M satisfies the first two conditions from 4.9.4, i.e. is of type Monte-Carlo. Moreover, it works in at most $2p(n)$ steps, hence it is a polynomial Monte-Carlo RTM. This proves that L belongs to $\mathcal{RP}$.

Since $\mathcal{ZPP}$ is closed under complements, every language from $\mathcal{ZPP}$ is also in co-$\mathcal{RP}$.

4.9.13 Theorem. We have

$$\mathcal{P} \subseteq \mathcal{ZPP}, \quad \mathcal{RP} \subseteq \mathcal{NP}, \quad \text{co-}\mathcal{RP} \subseteq \text{ co-}\mathcal{NP}.$$

The first inclusion is evident; indeed every deterministic Turing machine with polynomial complexity can be considered as a Las-Vegas RTM.

The second inclusion is less evident. The proof constructs, for a Monte Carlo RTM M, a non-deterministic Turing machine N such that $L(N)$ consists of all words for which there is a content of the random tape M that leads to an accepting state. Moreover, if M works in a polynomial time so does N.

The third inclusion is an easy consequence of the second one and the definitions of $\mathcal{RP}$ and co-$\mathcal{NP}$.

Chapter 5

Undecidability

5.1 Recursive and recursively enumerable languages

5.1.1 Recursively enumerable (RE) languages. A language L is said to be *recursively enumerable*, denoted *RE*, if there exists a Turing machine M that accepts L, i.e. for which $L = L(M)$.

In other words, there is a TM M such that for every $w \in L$ it accepts in state $q_f \in F$; for $w \notin L$ either M halts in non-accepting state or does not halt at all.

5.1.2 Recursive languages. A language L is said to be *recursive* if there exists a Turing machine M that decides L.

Note that a TM M decides a language L if and only if M halts on every input w, and if $w \in L$, then it halts in $q \in F$, and if $w \notin L$, it halts in $p \notin F$.

5.1.3 Remark. Languages that are not recursive are also called *algorithmically unsolvable* or *undecidable*. Similarly, we speak of undecidable or algorithmically unsolvable problems. Decision problems are usually called undecidable. The term algorithmically unsolvable is used usually for optimization problems.

Every recursive language is also RE, but note that the converse does not hold. We will show that there are languages that are RE but not recursive.

5.1.4 Proposition. If a language L is recursive then so is its complement $\overline{L}$.

5.1.5 Proposition. If a language L and its complement $\overline{L}$ are both RE, then they are recursive.

5.1.6 Proposition. Given a language L, one of the following possibilities occurs:

1. L and $\overline{L}$ are both recursive.
2. One of L and $\overline{L}$ is RE and the other is not RE.
3. L and $\overline{L}$ are not RE.

5.1.7 Code of a Turing machine. Every Turing machine M can be coded as a binary word. Indeed, denote parts of a TM M as follows: the set of states $Q = \{q_1, q_2, \dots, q_n\}$, the set of input symbols $\Sigma = \{0, 1\}$, the set of tape symbols $\Gamma = \{X_1, X_2, \dots, X_m\}$, where $X_1 = 0$, $X_2 = 1$ a $X_3 = B$, the initial state by q_1 and the final state q_2. Also, denote a rightward move D_1 and leftward move D_2. (It means $D_1 = R$ and $D_2 = L$.)

One entry of the transition function δ

$$\delta(q_i, X_j) = (q_k, X_l, D_r)$$

corresponds to the following word

$$t = 0^i 10^j 10^k 10^l 10^r.$$

A code of M, we denote it by $\langle M \rangle$, is then

$$\langle M \rangle = 111\, t_1\, 11\, t_2\, 11 \ldots 11\, t_p\, 111,$$

where $t_1, \ldots, t_p$ are words corresponding to all entries of the transition function of M.

5.1.8 A given binary word w can be enumerated in the following way: form $1w$ and regard it as a natural number written in binary. For instance, ϵ is the first word, 0 is second, 1 is third, etc., 100110 is $1100110 = 64 + 32 + 4 + 2 = 102$; in other words 100110 is the 102nd word in the numbering. In what follows, a binary word which is the i-th position is considered as w_i. Hence, $w_1 = \epsilon$, $w_{102} = 100110$, etc.

This method is an ordering by length and, for words of equal length, by lexicographic sorting.

5.1.9 Diagonal language L_d. First, let us make the following convention: If a binary word w does not have the form from 5.1.7 it is considered an encoding of Turing machine M with no transition, so it accepts no word (i.e. $L(M) = \emptyset$).

The language L_d consists of all binary words w such that the Turing machine encoded by w **does not accept** w. (Therefore, L_d also contains all words w that are not codes of any Turing machine, but it contains other words as well.)

5.1.10 Theorem. There is no Turing machine that accepts the diagonal language L_d. In other words, $L_d \neq L(M)$ for every Turing machine M.

Sketch of the proof. We proceed by contradiction. If there were a Turing machine M_0 such that $L_d = L(M_0)$, then M_0 has a code that is a binary word, i.e. $\langle M \rangle = w_i$ for a suitable i.

Then either w_i belongs or does not belong to L_d, but both possibilities lead to a contradiction.

Assume that $w_i \in L_d$. Then w_i satisfies the following condition: the Turing machine with code w_i does not accept w_i. But $L_d = L(M_0)$ where $w_i = \langle M_0 \rangle$ — a contradiction.

Assume that $w_i \notin L_d = L(M_0)$. Then the Turing machine with code w_i does not accept w_i. But the definition of L_d implies that w_i belongs to L_d — a contradiction.

Therefore, there is no Turing machine that accepts L_d.

5.1.11 The universal language. The *universal language* L_U is the set of all words of the form $\langle M \rangle \# w$ where $\langle M \rangle$ encodes a Turing machine that accepts the word $w \in \{0,1\}^\star$, i.e. $w \in L(M)$.

5.1.12 Universal Turing machine. We roughly describe a Turing machine that accepts the universal language L_U. (Such a Turing machine is called a *universal Turing machine* and we denote it by U.)

A universal Turing machine U has 4 tapes. The first tape contains the input, i.e. the word $\langle M \rangle \# w$, the second tape simulates the tape of the Turing machine M over w, and the third tape contains the code of the state in which M is. Further, U has an auxiliary tape; it is the fourth tape.

At the beginning, the Turing machine U has the input $\langle M \rangle \# w$ on its first tape, other tapes contain only blanks B. Recall that an encoded TM looks as follow:: Assume that M is $(Q, \{0,1\}, \{0,1,B\}, \delta, q_1, \{q_2\})$, where $Q = \{q_1, q_2, \ldots, q_n\}$. Denote 0 by X_1, 1 by X_2, and B by X_3, R will be D_1, L will be D_2. Then entries of the transition function $\delta(q_i, X_j) = (q_k, X_l, D_m)$ are encoded by

$$t = 0^i 10^j 10^k 10^l 10^m, \text{ where } 1 \leq i,k \leq n, 1 \leq j,l \leq 3, 1 \leq m \leq 2.$$

Turing machine M has the code

$$111\, t_1\, 11\, t_2\, 11 \ldots 11\, t_r\, 111.$$

The Turing machine U at first checks whether the input is a code of a Turing machine M followed by a binary word w. If not, U halts unsuccessfully.

If the input has the form $\langle M\rangle\#w$ for a TM M (which has some moves), U rewrites w on the second tape and it writes 0 on the third tape. The reason is that at the start of execution, M is in state q_1, which is coded as 0.

Now TM U simulates the steps of M in such a way that if M enters the state q_2 (final, "accepting" state of M), U halts successfully. This can be easily recognized: the third tape will have 00 surrounded by blanks.

Note that the construction of U needs further technical details; e.g. when the word w is copied on the second tape of U, the same coding of 0 and 1 is used as when forming a code of a word. It means that 0 is 10, 1 is 100. If blank of M is required on the second tape, it is written 1000.

5.1.13 Corollary. The universal language L_U is RE.

5.1.14 Proposition. L_U is not recursive.

Assume that L_U is recursive then there exists a TM M_1 which decides L_U. I.e. M always halts, successfully on words from L_U, unsuccessfully on words not belonging to L_U. Using M_1 we could decide the diagonal language L_d and we know that L_d is not even recursively enumerable, see 5.1.10.

5.1.15 Reductions. Recall the definition of a reduction from 4.3.1.

Given two decision problems $\mathcal{U}$ and $\mathcal{V}$, $\mathcal{U}$ *reduces* to $\mathcal{V}$ if there exists an algorithm (a program for RAM, a Turing machine) $\mathcal{A}$ that for every instance I of $\mathcal{U}$ constructs an instance I' of $\mathcal{V}$ in such a way that

$$I \text{ is YES instance of } \mathcal{U} \text{ iff } I' \text{ is YES instance of } \mathcal{V}.$$

Denote a reduction of $\mathcal{U}$ to $\mathcal{V}$ as

$$\mathcal{U} \lhd \mathcal{V}.$$

This definition is important for languages as well. A decision problem is viewed as the language of all words that are YES instances.

5.1.16 Proposition. Given two decision problems $\mathcal{U}$ and $\mathcal{V}$ such that $\mathcal{U} \lhd \mathcal{V}$, we have:

1. $\mathcal{U}$ undecidable implies $\mathcal{V}$ undecidable.
2. $\mathcal{U}$ not in RE implies $\mathcal{V}$ not in RE.
3. $\mathcal{V}$ recursive implies $\mathcal{U}$ recursive.

5.1.17 Proposition. Given two languages

$$L_e = \{M \mid L(M) = \emptyset\}, \quad L_{ne} = \{M \mid L(M) \neq \emptyset\},$$

L_{ne} is RE but not recursive and L_e is not RE.

5.1.18 Remark. Notice that L_e is the complement of L_{ne}. Indeed, if a word w doess not encode any Turing machine, then it encodes a TM with no moves, so it belongs to L_e.

We can use the universal Turing machine U for showing that L_{ne} is RE. From the reduction $L_U \lhd L_{ne}$ and 5.1.16, we get L_{ne} is not recursive. The fact that L_e is not RE then follows from 5.1.6.

5.1.19 Theorem (Rice). Any nontrivial property of RE languages is undecidable.

By a nontrivial property we mean a property that some RE language has and some RE language does not have.

5.2 Other undecidable problems

In this section we introduce further problems/languages that are undecidable. Note that here, the universal language L_{UN} plays similar role for undecidable languages as the problem (and corresponding language) SAT plays for $\mathcal{NP}$ complete problems/languages.

The first undecidable problem is Post correspondence problem.

5.2.1 Post Correspondence Problem (PCP). Given two lists of words A, B over an alphabet Σ

$$A = (w_1, w_2, \dots, w_k), \quad B = (x_1, x_2, \dots, x_k),$$

where $w_i, x_i \in \Sigma^\star$, $i = 1, 2, \dots, k$, we say that A, B *has a solution* if there exists a finite sequence $i_1, i_2, \dots, i_r$ of indexes $i_j \in \{1, 2, \dots, k\}$ such that

$$w_{i_1}\, w_{i_2}\, \dots\, w_{i_r} = x_{i_1}\, x_{i_2}\, \dots\, x_{i_r}.$$

Question: Is there a solution of a given instance?

5.2.2 Examples.

1. Given two lists

	1	2	3	4	5
A	011	0	101	1010	010
B	1101	00	01	00	0

This instance has a solution, for example $2, 1, 1, 4, 1, 5$ gives

$$w_2\, w_1\, w_1\, w_4\, w_1\, w_5 = 00110111010011010 = x_2\, x_1\, x_1\, x_4\, x_1\, x_5.$$

2. Given two lists of binary words

	1	2	3	4	5
A	11	0	101	1010	010
B	101	00	01	00	0

This instance does not have a solution.

5.2.3 Modified Post Correspondence Problem (MPCP). Given two lists of words A, B over a given Σ.

$$A = (w_1, w_2, \dots, w_k), \quad B = (x_1, x_2, \dots, x_k),$$

where $w_i, x_i \in \Sigma^\star$, $i = 1, 2, \dots, k$, we say that A, B *has a solution* if there exists a sequence of indices $1, i_1, i_2, \dots, i_r$, i.e. $i_j \in \{1, 2, \dots, k\}$ such that

$$w_1\, w_{i_1}\, w_{i_2}\, \dots\, w_{i_r} = x_1\, x_{i_1}\, x_{i_2}\, \dots\, x_{i_r}.$$

Question: Is there a solution of a given instance?

5.2.4 Remark. MPCP differs from PCP by the fact that in MPCM we require that the solution must begin with the index 1. The importance of MPCP can be seen from the following theorem.

5.2.5 Theorem. We have

$$L_u \lhd \text{ MPCP } \lhd \text{ PCP}.$$

5.2.6 Theorem. We have

$$L_u \lhd \text{ MPCP } \lhd \text{ PCP}.$$

Sketch of the second reduction. Given an instance of MPCP

$$A = (w_1, w_2, \ldots, w_k), \quad B = (x_1, x_2, \ldots, x_k),$$

Assume that $\#$ and $*$ do not belong to Σ, Let us construct a new instance of PCP by:

$$C = (y_0, y_1, \ldots, y_k, y_{k+1}), \quad D = (z_0, z_1, \ldots, z_k, z_{k+1}),$$

where

1. for every $i = 1, \ldots, k$ the word y_i was created from w_i in such a way that **behind** every symbol of w_i we placed $*$; analogously, z_i was created from x_i by adding $*$ **in front of** every symbol of x_i.
2. $y_0 = *y_1$; $z_0 = z_1$.
3. $y_{k+1} = *\#$, $z_{k+1} = \#$.

It is not difficult to see that A, B has a solution $1, i_1, \ldots, i_r$ if and only if C, D has a solution, and the solution of C, D must be $0, i_1, \ldots, i_r, k+1$.

The first reduction is more difficult. It consists of a description of the work of a Turing machine M over a word w by two lists of words.

5.2.7 Corollary. PCP is undecidable.

5.2.8 Remark. If we restrict the length of a sequence of indexes that forms a solution $i_1, i_2, \ldots, i_r$, (i.e. we restrict value r), the problem becomes algorithmically solvable — there is a "brute force" algorithm that produces systemically sequences $i_1, i_2, \ldots, i_s$ and for every such sequence it checks whether it is a solution.

At the same time, if A and B are sets (instead of lists) then the problem is polynomially solvable.

5.2.9 Ambiguous Context Free Grammars. Given a context free grammar $\mathcal{G} = (N, \Sigma, S, P)$, where N is the set of nonterminal symbols, Σ the set of terminal symbols, S is a start symbol, and P a set of production rules of the type $X \to \alpha$ for $X \in N$, $\alpha \in (N \cup \Sigma)^\star$.

Question: Is there a word w with two different derivation (parse) trees?

5.2.10 Theorem. We have

$$\text{PCP } \lhd \text{ Problem whether a given CF grammar is ambiguous.}$$

5.2.11 Idea of the proof of 5.2.10. Given an instance of PCP, i.e. two lists $A = (w_1, w_2, \ldots, w_k)$ and $B = (x_1, x_2, \ldots, x_k)$. We construct a CF grammar $\mathcal{G} = (\{S, A, B\}, \Sigma \cup \{a_1, a_2, \ldots, a_k\}, S, P)$ where P contains the following rules

$$S \to A \mid B,$$

$$A \to w_1\, A\, a_1 \mid w_2\, A\, a_2 \mid \ldots | w_k\, A\, a_k,$$

$$A \to w_1\, a_1 \mid w_2\, a_2 \mid \ldots | w_k\, a_k,$$

$$B \to x_1\, B\, a_1 \mid x_2\, B\, a_2 \mid \ldots | x_k\, B\, a_k,$$

$$B \to x_1\, a_1 \mid x_2\, a_2 \mid \ldots | x_k\, a_k,$$

$\mathcal{G}$ is ambiguous if and only if there is a word $wa_{i_1}a_{i_2}\ldots a_{i_r}$, $w \in \Sigma^\star$, with two different derivation trees. And this occurs if and only if the pair A, B has a solution. (Note that two different derivation trees for a word u exist only when $u = wa_{i_1}a_{i_2}\ldots a_{i_r}$ and the first rule used for the two derivations are $S \to A$ for the first derivation tree and $S \to B$ for the second derivation tree. Hence w can be formed from A and from B using the same sequence of indexes.)

5.2.12 Theorem. Given CF grammars $\mathcal{G}_1$ and $\mathcal{G}_2$. Denote $L(\mathcal{G}_1)$ and $L(\mathcal{G}_2)$ languages generated by $\mathcal{G}_1$ and $\mathcal{G}_2$. The following problems are undecidable.

1. $L(\mathcal{G}_1) \cap L(\mathcal{G}_2) = \emptyset$.
2. $L(\mathcal{G}_1) = L(\mathcal{G}_2)$.
3. $L(\mathcal{G}_1) \subseteq L(\mathcal{G}_2)$.
4. $L(\mathcal{G}_1) = \Sigma^\star$.