Data structures and algorithms
Glossary of definitions and theorems

English version by Viktória Vadon
based on [1, 2]

Latest version: April 18, 2025

Contents

1 Introduction, mathematics and representation of numbers

Definition 1.1 (The greatest integer (floor) function)**.** The greatest integer function assigns an integer k to every real number x, that is the greatest of all integers not greater than x. In other words, k is the *closest* integer that isn't greater than x. Formally:

$$\lfloor x \rfloor = \max\{k \in \mathbb{Z} \mid k \leq x\}, \tag{1}$$

in other words, k is the unique integer such that $k \leq x < k + 1$.

It is also known as the *integer part or integral part function*, denoted by $[x]$.

Definition 1.2 (The least integer (ceil) function)**.** The least integer function assigns an integer k to every real number x, that is the smallest of all integers not smaller than x. In other words, k is the *closest* integer that isn't smaller than x. Formally:

$$\lceil x \rceil = \min\{k \in \mathbb{Z} \mid k \geq x\}, \tag{2}$$

in other words, k is the unique integer such that $k - 1 < x \leq k$.

Definition 1.3 (The rounding function)**.** The rounding function assigns the closest integer k to every real number x. If the closest integer is not unique (when the decimal part of the fraction is .5), the greater neighbor is chosen. Formally:

$$\text{Round}(x) = \left\lfloor x + \frac{1}{2} \right\rfloor \tag{3}$$

Definition 1.4 (The fraction function or fractional part function)**.** To every real number x, the fraction function assigns a real number $\{x\}$, which shows how much x is greater than its integer part. Formally,

$$\{x\} = x - \lfloor x \rfloor. \tag{4}$$

The fractional part always satisfies $0 \leq \{x\} < 1$.

Definition 1.5 (Whole quotient, div operation)**.** Let a and b be integers. We define the whole quotient (result of whole division) as

$$a \text{ div } b := \begin{cases} \left\lfloor \frac{a}{b} \right\rfloor & \text{if } b \neq 0, \\ \text{not defined} & \text{if } b = 0. \end{cases} \tag{5}$$

Definition 1.6 (Whole remainder, mod operation)**.** Let a and b be integers. We define the whole remainder as

$$a \text{ mod } b := \begin{cases} a - (a \text{ div } b) \cdot b = a - \left\lfloor \frac{a}{b} \right\rfloor \cdot b, & \text{if } b \neq 0, \\ a & \text{if } b = 0. \end{cases} \tag{6}$$

By convention, $x \text{ mod } 1 := \{x\}$, and is defined for every real number x.

Theorem 1.7 (Number of digits)**.** Suppose x is an integer (suppose it is given in base ten) that we wish to write in base b, that is, in the form

$$x = c_n c_{n-1} \ldots c_1 c_{0_{(b)}}. \tag{7}$$

Then the number of required digits is $n + 1 = \lfloor \log_b x \rfloor + 1$.

2 Algorithms: representation and quantification

2.1 Data

Definition 2.1 (Data)**.** A (relevant) piece of information or detail used to qualify or quantify someone or something.

Definition 2.2 (Abstract data)**.** Abstract data is an element of a specified, feasible set. The feasible set contains all possible values (that may be used to quantify an object and that we may use in calculations) according to the mathematical model.

Definition 2.3 (Abstract data type)**.** Abstract data type consists of the feasible set of abstract data and the set of operations defined on this set.

Definition 2.4 (Data structure)**.** A specific, complete implementation of an abstract data type, including the implementation of the data as well operations on it.

2.2 Program, algorithm

Definition 2.5 (Program)**.** A sequence of instructions for the computer to execute.

Definition 2.6 (Procedure)**.** A unit of a program for a specified task.

Definition 2.7 (Algorithm)**.** A specified, step-by-step calculation method, a tool for solving computation problems.

Definition 2.8 (Recursion)**.** An algorithm is recursive, if it calls itself with different (smaller) input.

Definition 2.9 (Divide and conquer principle)**.** A principle of planning algorithms, that creates the solution in the following steps:

- divide: divide the problem into independent, smaller subproblems of the same type.
- conquer: solve the subproblems, often recursively; handle a basic case separately.
- unite: combine partial solutions to solve the original big problem.

Definition 2.10 (Dynamic programming)**.** A principle of planning algorithms. It creates a sequence of (not independent) subproblems that build on each other and can be solved in sequence, using the previous results. It is often used in optimization problems.

Definition 2.11 (Input size, problem size)**.** Let A be (an implementation of) an algorithm. Let D be the set of possible inputs, and $x \in D$ a given input. We define the input size $|x|$ as the number of bits x is stored on, given a specific data structure. We always have $|x| \in \mathbb{N}$.

Definition 2.12 (Time and storage requirement)**.** Again, let A be (an implementation of) an algorithm, D be the set of possible inputs, and $x \in D$ a given input. We denote by $t_A(x)$ the *time requirement* (in number of operations required) when A is run on input x. Analogously, we denote by $s_A(x)$ the *storage requirement* (in number of bytes occupied in memory) when A is run on input x.

Definition 2.13 (Time complexity)**.** Again, let A be (an implementation of) an algorithm, D be the set of possible inputs. We define the *time complexity* of A as

$$T_A(n) := \max_{\substack{x \in D \\ |x| \leq n}} t_A(x), \tag{8}$$

that is, the largest possible runtime on inputs of size at most n.

Definition 2.14 (Storage complexity)**.** Again, let A be (an implementation of) an algorithm, D be the set of possible inputs. We define the *storage complexity* of A as

$$S_A(n) := \max_{\substack{x \in D \\ |x| \leq n}} s_A(x), \tag{9}$$

that is, the largest possible storage required for inputs of size at most n.

3 Growth rates

Definition 3.1 (Growth rates, *order* of functions)**.** Note: I will not ask the entire definition, but the notations individually, e.g. "define the small O notation".

Let $f : \mathbb{N} \to \mathbb{R}^+$ be a function, we may describe its growth rate as some of the following:

1. *big O notation.* Se say that $f(n) = O\big(g(n)\big)$, "$f(n)$ is *big 'O' (of)* $g(n)$", if there exists a constant $c > 0$ and a threshold $n_0 \in \mathbb{N}$ such that for all $n \geq n_0$, $f(n) \leq c \cdot g(n)$, in other words, $\frac{f(n)}{g(n)} \leq c$, the sequence of fractions is *bounded from above.*

2. *small/little O notation.* Se say that $f(n) = o\big(g(n)\big)$, "$f(n)$ is *small/little 'O' (of)* $g(n)$", if for all constants $c > 0$, there exists a threshold $n_0 \in \mathbb{N}$ such that for all $n \geq n_0$, $f(n) \leq c \cdot g(n)$, in other words, $\frac{f(n)}{g(n)} \to 0$ as $n \to \infty$.

3. *big Omega notation (Knuth).* Se say that $f(n) = \Omega\big(g(n)\big)$, "$f(n)$ is *big 'Omega' (of)* $g(n)$", if there exists a constant $c > 0$ and a threshold $n_0 \in \mathbb{N}$ such that for all $n \geq n_0$, $f(n) \geq c \cdot g(n)$, in other words, $\frac{f(n)}{g(n)} \geq c$, the sequence of fractions is *bounded from below.*

4. *small/little Omega notation.* Se say that $f(n) = \omega\big(g(n)\big)$, "$f(n)$ is *small/little 'Omega' (of)* $g(n)$", if for all constants $c > 0$, there exists a threshold $n_0 \in \mathbb{N}$ such that for all $n \geq n_0$, $f(n) \geq c \cdot g(n)$, in other words, $\frac{f(n)}{g(n)} \to \infty$ as $n \to \infty$.

5. *(big) Theta notation.* Se say that $f(n) = \Theta\big(g(n)\big)$, "$f(n)$ is *(big) 'Theta' (of)* $g(n)$", if there exist constants $0 < c_1 < c_2$ and a threshold $n_0 \in \mathbb{N}$ such that for all $n \geq n_0$, $c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$, in other words, $c_1 \leq \frac{f(n)}{g(n)} \leq c_2$, the sequence of fractions is *bounded.*

Definition 3.2 (Some common growth rates)**.** Note: I will not ask you to recite the entire table, instead the names individually, e.g. "what is a polynomial growth rate?".

constant	$f(n) = \Theta\big(1\big)$
linear	$f(n) = \Theta\big(n\big)$
quadratic	$f(n) = \Theta\big(n^2\big)$
cubic	$f(n) = \Theta\big(n^3\big)$
polynomial	$f(n) = \Theta\big(n^k\big)$, for some $k \in \mathbb{R}^+$
logarithmic	$f(n) = \Theta\big(\log(n)\big)$
exponential	$f(n) = \Theta\big(a^n\big)$, for some $a > 1$

Definition 3.3 (Polynomially faster growth)**.** We say that $f : \mathbb{N} \to \mathbb{R}^+$ grows polinomially faster than n^p, for $p \geq 0$, if there exists $\varepsilon \in \mathbb{R}^+$ such that $f(n) = \Omega\big(n^{p+\varepsilon}\big)$.

Definition 3.4 (Polynomially slower growth)**.** We say that $f : \mathbb{N} \to \mathbb{R}^+$ grows polinomially slower than n^p, for $p \geq 0$, if there exists $\varepsilon \in \mathbb{R}^+$ such that $f(n) = O\big(n^{p-\varepsilon}\big)$.

Definition 3.5 (Recurrence equation)**.** A recursive equation is a functional equation of some unknown function $T : \mathbb{N} \to \mathbb{R}^+$, where $T(n)$ is given in terms of one or more $T(k_i)$, with $k_i < n$.

Theorem 3.6 (The "master" theorem)**.** Suppose we have a recursive equation

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

where $T : \mathbb{N} \to \mathbb{R}^+$ is the unknown function, $f : \mathbb{N} \to \mathbb{R}^+$ is a known function, $a \geq 1$ and $b > 1$ are known constants. (The theorem also holds with $\lfloor \frac{n}{b} \rfloor$ or $\lceil \frac{n}{b} \rceil$ in the place of $\frac{n}{b}$.)

Define $p := \log_b(a)$ and the so-called *test polynomial* $g(n) := n^p$. Under specific conditions, we can determine the growth rate of $T(n)$:

1. If $f(n)$ grows *polynomially slower* than $g(n)$, then

$$T(n) = \Theta\big(g(n)\big). \tag{10}$$

2. If $f(n) = \Theta\big(g(n)\big)$, then

$$T(n) = \Theta\big(g(n) \cdot \log(n)\big). \tag{11}$$

3. Suppose $f(n)$ grows *polynomially faster* than $g(n)$, as well as the so-called *regularity condition* holds for f, that is,

$$\text{exists } c < 1, c \in \mathbb{R}^+ \text{ and } n_0 \in \mathbb{N} \text{ such that for all } n \geq n_0, a \cdot f\left(\frac{n}{b}\right) \leq c \cdot f(n). \tag{12}$$

If both above conditions hold, then

$$T(n) = \Theta\big(f(n)\big). \tag{13}$$

3.1 Fibonacci numbers

Definition 3.7 (Fibonacci numbers)**.** The Fibonacci numbers is a number series defined recursively as

$$\left.\begin{array}{r} F_0 := 0 \\ F_1 := 1 \end{array}\right\} \quad \text{initial conditions}$$
$$n \geq 2: \; F_n := F_{n-1} + F_{n-2} \quad \text{recursive condition} \tag{14}$$

Theorem 3.8 (Binet's formula)**.** For any $n \in]N$, element F_n of the Fibonacci series can be obtained directly by the formula

$$F_n = \frac{1}{\sqrt{5}}\left(\Phi^n - \overline{\Phi}^n\right), \tag{15}$$

where

$$\Phi = \frac{1+\sqrt{5}}{2} \approx 1.618 \qquad \overline{\Phi} = \frac{1-\sqrt{5}}{2} \approx -0.618$$

Theorem 3.9 (Fibonacci numbers with rounding)**.** Based on Binet's formula, for any $n \in \mathbb{N}$, the nth Fibonacci number F_n can be obtained as

$$F_n = \text{Round}\left(\frac{1}{\sqrt{5}}\Phi^n\right). \tag{16}$$

4 Algorithms from number theory

4.1 Divisibility, greatest common divisor

Definition 4.1 (Divisibility)**.** For integers d and a, we say that d (wholly) divides a and write $d|a$, if there exists an integer k such that $k \cdot d = a$. We may also say that d is a divisor of a or that a is a multiple of d.

Definition 4.2 (Prime number)**.** We call an integer $p > 1$ a prime if its only positive divisors are 1 and p itself.

Theorem 4.3 (Whole division with remainder)**.** Let $a \in \mathbb{Z}$ and $n \in \mathbb{Z}^+$. Then there exist a unique pair of $q, r \in \mathbb{Z}$ such that $0 \leq r < n$ that satisfy

$$a = q \cdot n + r.$$

We call q the quotient and r the remainder. They are also given by $q = a \text{ div } n$, $r = a \bmod n$.

Definition 4.4 (Common divisor)**.** We say that $d \in \mathbb{Z}$ is a common divisor of $a, b \in \mathbb{Z}$, if it is a divisor of both, i.e., $d|a$ and $d|b$.

Definition 4.5 (Linear combination)**.** We say that $s \in \mathbb{Z}$ is a linear combination of $a, b \in \mathbb{Z}$, if there exist $x, y \in \mathbb{Z}$ such that $s = x \cdot a + y \cdot b$. We call x and y the coefficients of the linear combination. We denote by $L(a, b)$ the set of all (numbers that are) linear combinations of a and b.

Theorem 4.6 (Properties of the (common) divisor)**.** Let $d \in \mathbb{Z}$ be a common divisor of $a, b \in \mathbb{Z}$. Then

1. if both $d|a$ and $a|d$, necessarily $d = \pm a$.
2. d is also a divisor of any linear combination of a and b, i.e., for any $s \in L(a, b)$, $d|s$.

Definition 4.7 (The greatest common divisor)**.** For $a, b \in \mathbb{Z}$, we define their greatest common divisor as follows:

$$d^\star = \gcd(a, b) := \begin{cases} 0, & \text{if } a = b = 0, \\ \max\limits_{\substack{d|a \\ d|b}} d, & \text{otherwise.} \end{cases} \tag{17}$$

Definition 4.8 (Relative primes)**.** We say that $a, b \in \mathbb{Z}$ are relative primes if $\gcd(a, b) = 1$.

Theorem 4.9 (Elementary properties of the greatest common divisor)**.** Let $a, b \in \mathbb{Z}$ and $d^\star = \gcd(a, b)$. Then

1. $\gcd(a, b) = \gcd(b, a)$
2. $\gcd(a, 0) = a$.
3. the greatest common divisor divides all linear combinations of a and b, i.e., $d^\star|s$ for all $s \in L(a, b)$.

Theorem 4.10 (Representation of the greatest common divisor)**.** Let $a, b \in \mathbb{Z}$, and suppose not both are 0. Then their greatest common divisor is equal to their smallest positive linear combination. In formula:

$$d^\star = \gcd(a, b) = \min_{\substack{s \in L(a,b) \\ s>0}} s =: s^\star = x^\star \cdot a + y^\star \cdot b, \tag{18}$$

where we also introduce the notation $x^\star, y^\star$ for the coefficients of $s^\star$.

Theorem 4.11 (Description of the set of linear combinations)**.** Let $d^\star := \gcd(a, b)$, and the set of its multiples $M := \{k \cdot d^\star, k \in \mathbb{Z}\}$. Then

$$L(a, b) \equiv M.$$

In words: all linear combinations of a and b are multiples of $d^\star$, and vica versa.

Theorem 4.12 (Reduction theorem (of the greatest common divisor))**.** For any $a, b \in \mathbb{Z}$,

$$\gcd(a, b) = \gcd(a - b, b).$$

Theorem 4.13 (Recursion theorem (of the greatest common divisor))**.** Let $n, a \in \mathbb{Z}$, then

$$\gcd(a, b) = \gcd(b, a \mod b).$$

Theorem 4.14 (Lamé)**.** Suppose the input for the recursive Euclidean algorithm are $a, b \in \mathbb{N}, a \geq b$, and suppose $b < F_{k+1}$ for some Fibonacci number (see Def 3.7). Then the number of recursive calls is less than k.

4.2 Congruence, linear congruence equation, multiplicative inverse

Definition 4.15 (Congruence)**.** For $a, b \in \mathbb{Z}$ and $n \in \mathbb{Z}, n \neq 0$, we say that a and b are congruent modulo n and write $a \equiv b \mod n$, if $n|(a - b)$, or equivalently, if $(a \mod n) = (b \mod n)$ (in words, a and b have the same remainder when divided by n).

Definition 4.16 (Linear congruence equation)**.** We call the equation

$$a \cdot x \equiv b \mod n, \tag{19}$$

with known constants $a, b \in \mathbb{Z}$, $n \in \mathbb{Z}^+$, and unknown $x \in \mathbb{Z}$, the linear congruence equation.

Theorem 4.17 (Solvability of the linear congruence equation)**.** Consider the linear congruence equation (19), and let $d^\star = \gcd(a, n) = x^\star \cdot a + y^\star \cdot n$.

1. If $d^\star \not| b$, the linear congruence equation has no solution.
2. If $d^\star | b$, the linear congruence equation has infinitely many solutions, however all of them can be obtained from a system of $d^\star$ many *incongruent* solutions in $[0, n)$, by adding multiples of n. The incongruent solutions are:

$$\begin{aligned} x_0 &= x^\star \cdot \frac{b}{d^\star} \mod n, \\ x_i &= x_0 + i \cdot \frac{n}{d^\star} \mod n \\ &= x_{i-1} + \frac{n}{d^\star} \mod n, \qquad \text{for } i = 1, \ldots, d^\star - 1. \end{aligned} \tag{20}$$

Definition 4.18 (Multiplicative inverse)**.** Let $a \in \mathbb{Z}$, $n \in \mathbb{Z}^+$ such that $\gcd(n, a) = 1$, and consider the linear congruence equation

$$ax \equiv 1 \mod n.$$

Given $\gcd(n, a) = 1$, the equation has a single solution x_0 in $[0, n)$. We call this the *multiplicative inverse* of a modulo n and denote $x_0 = a^{-1} \mod n$.

Theorem 4.19 (Fermat's little theorem)**.** If p is a prime number, then for all $a = 1, \ldots, p - 1$,

$$a^{p-1} \equiv 1 \mod p.$$

5 Dynamic sets

Definition 5.1 (Dynamic set)**.** A dynamic set is a dataset that changes (elements are added, removed, modified) during the run of the algorithm using it.

Definition 5.2 (Sequence)**.** A sequence is a data structure where elements/items/records are stored in a linear order (whether physically, or as defined by the operations of the sequence). Typical operations are: search, insert, delete.

Definition 5.3 (Array)**.** An array is a data structure implementing the sequence that consists of consecutive memory bins. Each bin may store an individual data record, and is directly accessible by its index. The array has attributes head, end, length and arraysize. It supports operations search, insert and delete.

Definition 5.4 (Linked list)**.** A linked list is a data structure implementing the sequence. Records may be stored separately, but each element contains a pointer to the next, which establishes the linear order. It has attributes head, sometimes end. It supports operations search, insert, delete.

Definition 5.5 (Queue (data structure))**.** The queue is a dynamic set where insertion and deletion may only happen at predefined points. The inserted element is the newest, and we always delete the oldest element. Its supported operations are insert (push) and delete (pop).

Definition 5.6 (Stack (data structure))**.** The stack is a dynamic set where insertion and deletion may only happen at predefined points. We always remove the latest inserted element. Supported operations are push (insert) and pop (remove).

Definition 5.7 (Hash table)**.** The hash table is a dynamic set, but not a sequence. The table is allocated a continuous chunk of memory, and rows are accessible directly by index. Data records are assigned a row according to the so-called hash function of the key field. It supports operations search, insert and delete.

Definition 5.8 (Conflict (in hash table))**.** If the hash function assigns the same value/row to two data records with different keys, we call it a conflict.

Definition 5.9 (Open address hash table)**.** The hash function takes two variables, the key k and the index of the trial $t = 0, 1, \ldots, N-1$. The sequence $h(k,0), h(k,1), \ldots, h(k,N-1)$ is called the *search sequence*, and must contain all row indices $i = 0, 1, \ldots, N-1$ in some order.

Definition 5.10 (Cluster (in hash table))**.** A group of consecutive occupied rows in a hash table is called a cluster. The size of the cluster is the number of the rows in question.

6 Selection problem and sorting

Definition 6.1 (The "selection problem")**.** Suppose a set A of n different numbers, and a rank $1 \leq k \leq n$ are given. The task is to find the kth smallest element, that is, element $x \in A$ so that exactly $k-1$ elements of A are smaller than x.

Definition 6.2 (Median)**.** The *median* of a dataset (of numbers) is the middle element of the (increasingly) *sorted* dataset. If the dataset has an odd number $n = 2k+1$ of elements, the median has rank $k+1$ in the sorted set. If the dataset has an even number $n = 2k$ of elements, there are two middle elements, rank k and $k+1$. (We call them the lower and the upper median.)

Definition 6.3 (Descartes-product)**.** For sets A and B, we define their Descarted-product as the set

$$A \times B = \{(a,b), a \in A, b \in B\}$$

containing all possible ordered pairs of elements.

Definition 6.4 (Relation)**.** Let A be a set. A relation on set A is a subset $\varrho \subseteq A \times A$ (ϱ is the Greek letter "rho"). We say that $a, b \in A$ are in ϱ-relation if $(a,b) \in \varrho$, and for short write $a \varrho b$.

Definition 6.5 (Order relation (ordering))**.** We say that the relation $\varrho \subseteq a \times A$ is an *order relation (ordering)* on A if:

a) it is *reflexive*: $a \varrho a$ for all $a \in A$,

b) it is *transitive*: if $a \varrho b$ and $b \varrho c$, then also $a \varrho c$,

c) it is *antisymmetric*: if $a \varrho b$ and $b \varrho a$, then $a = b$.

We say that ϱ is a complete or linear ordering, if on top of the above, also

d*) at least one of $a \varrho b$ and $b \varrho a$ holds for any $a, b, \in A$.

Definition 6.6 (Some properties that may be applied to sort algorithms)**.** Note: I will not ask the entire definition, but individual properties, e.g. "what does it mean for a sorting algorithm to be stable?"

a) *sort in place*: the result replaces the original data, extra storage use is at most $O(1)$.

b) *adaptivity*: the algorithm is able to exploit any pre-existing order among the keys (and reduce its runtime accordingly).

c) *stability*: preserve original order of records with the same sorting key.

d) *comparison-based*: the algorithm is based on comparing keys to other keys (as opposed to grouping keys by their values).

Theorem 6.7 (Lower bound on comparison-based sorting)**.** Any *comparison-based* sorting algorithm must use at least $\Omega(n \log(n))$ comparisons.

Theorem 6.8 (Stirling's formula)**.** For $n \geq 3$, the factorial satisfies:

$$\frac{n^n}{e^n} < n! < \frac{(n+1)^{n+1}}{e^n}.$$

6.1 Runtime and important properties of common sorting algorithms

Note: I won't ask you to recite this whole table. Instead I may ask questions like:

- "name 3 comparison-based sorting algorithms"
- "what is the fastest comparison-based sorting algorithm you know?"
- "what is the runtime and memory requirement of merge sort?"

algorithm	time complexity	storage requirements	remarks
insertion sort	$O(n^2)$	sort in place	
(minimum) selection sort	$O(n^2)$	sort in place	many comparisons, but few elements moved
quicksort	$O(n \log(n)) \sim O(n^2)$	sort in place	
merge sort	$O(n \log(n))$	$O(n)$	can be used for externally stored data
heap sort	$O(n \log(n))$	sort in place	
counting sort / binsort	$O(n+m)$	$O(n+m)$	not comparison-based! m is the number of possible values

7 Graphs and trees

7.1 Huffman code

Definition 7.1 (Code (encoding))**.** A custom character table; each character is given a (different) bit sequence, called its code.

Definition 7.2 (Prefix code)**.** A code that may have varying code lengths, but no code is a continuation of another; conversely, no code is included as the beginning of another.

7.2 Introduction to graphs

Definition 7.3 (Graph (simple, undirected graph))**.** A graph is a pair $G = (V, E)$, where V is a finite set called the vertices, and E is a set of *unordered* pairs $e = \{u, v\}$ (with $u \neq v$) from V, called the edges.

Definition 7.4 (Directed graph (digraph))**.** A directed graph is a pair $G = (V, E)$, where V is a finite set and E is a set of *ordered* pairs $e = (u, v)$ (with $u \neq v$) of V ($E \subseteq V \times V$).

Definition 7.5 (Loop)**.** A *loop* is an edge $e = (v, v)$, connecting a vertex to itself.

Definition 7.6 (Weighted graph (network))**.** A graph (directed or undirected) becomes a weighted graph or network, when we assign a number, called *weight*, to each of its edges.

Definition 7.7 (Walk, trail, path; circuit, cycle)**.**

- A *walk* in a graph is a sequence of connecting edges: $(v_0, v_1), (v_1, v_2), (v_2, v_3), \ldots, (v_{n-1}, v_n)$. It is possible to repeat both vertices and edges.
- A *trail* is a special walk where no edges are repeated, but vertices may be repeated.
- A *path* is a special trail where no vertices are repeated.
- A *circuit* is a closed trail: that ends in the same vertex where it started.
- A *cycle* is a closed path: the end vertex is the same as the start vertex.

Definition 7.8 (Length of a path)**.**

- In an unweighted graph, the length of a path is the number of edges along the path.
- In a weighted graph, the *length* of a path is the sum of weights of all edges alongside the path. Formally, if $G = (V, E)$, $w : E \to \mathbb{R}$, and $P = (e_1, e_2, \ldots, e_k)$, then $w(P) := \sum_{i=1}^{k} w(e_k)$.

Definition 7.9 (Distance)**.** The distance of vertices u and v, denoted by $\delta(u, v)$, is the length of the shortest path between them, or ∞ if there is no path between them. Any path P with length $\delta(u, v)$ is a shortest path.

Definition 7.10 (Neighbor)**.**

- In an undirected graph, we say that vertices u and v are neighbors if $\{u, v\} \in E$.
- In a directed graph, v is an out-neighbor of u if $(u, v) \in E$, and an in-neighbor of u if $(v, u) \in E$. Together, we may refer to in- and out-neighbors simply as neighbors.

Definition 7.11 (Degree)**.**

- In an undirected graph, a degree of a vertex is the number of its neighbors.
- In a directed graph, the out-degree of a vertex is the number of its out-neighbors, analogously for in-degree. Its (total) degree is the sum of these two.

Definition 7.12 (Adjacency matrix)**.**

- For a directed graph on n vertices, the adjacency matrix Adj is an $n \times n$ matrix, both its rows and columns indexed by the vertices. Its entries are

$$A_{u,v} = \begin{cases} 1, & \text{if } (u, v) \in E, \\ 0, & \text{otherwise.} \end{cases}$$

- For a weighted, directed graph on n vertices, the adjacency matrix Adj is an $n \times n$ matrix, both its rows and columns indexed by the vertices. Its entries are

$$A_{u,v} = \begin{cases} w(u, v), & \text{if } (u, v) \in E, \\ \text{NIL}, & \text{otherwise.} \end{cases}$$

7.3 Introduction to trees

Definition 7.13 (Connectivity (in undirected graphs))**.**

a) We say that vertices u and v are connected if there is a path between them.
b) We say that the graph G is connected if any two vertices are connected.
c) We call the set of all vertices connected to a given vertex v the connected component of v, and a connected component of the graph.

Definition 7.14 (Forest)**.** A *forest* is an acyclic graph: a graph with no cycle.

Definition 7.15 (Equivalent definitions of a tree graph)**.** A tree is:

- a connected, acyclic graph;
- a connected graph with the least possible edges;
- a cycle-free graph with the most possible edges.

Definition 7.16 (Leaf)**.** A vertex with degree one (a single neighbor) in a tree is called a leaf.

Definition 7.17 (Rooted tree)**.** A rooted tree is a tree with a distinguished vertex called the *root*.

Definition 7.18 (Generation, height)**.** In a rooted tree, we call the distance of a vertex from the root its generation. The last non-empty generation in a rooted tree is called its height.

Definition 7.19 (Child, parent)**.** In a rooted tree, "neighbors" turn into a directed relation: the neighbor in the higher/earlier generation (drawn above) is called the parent, and the neighbor in the lower/later generation is called the child. Children of the same parent vertex are called siblings.

Definition 7.20 (Ancestor, descendant)**.** Suppose the path from a vertex u to the root crosses vertex v. Then we call v an ancestor of u, and u a descendant of v.

Definition 7.21 (Subtree)**.** The subtree of a vertex u consists of all its descendants. We can think of it as a rooted tree with root u.

Definition 7.22 (Binary tree)**.** A binary tree is a rooted tree where all vertices have at most 2 children (the root has degree at most 2 and all other vertices have degree at most 3). We distinguish the children as left child and right child.

Definition 7.23 (Complete binary tree)**.** A binary tree is complete, if it is being built/filled up generation by generation, and left to right.

Definition 7.24 (Heap (max-heap))**.** A max-heap is a data structure based on a complete binary tree, that satisfies the *max-heap property*: the key stored in any vertex is *larger* then the key in any of its children.

7.4 Binary search tree

Definition 7.25 (Binary search tree)**.** A binary search tree is a binary tree that satisfies the so-called *binary search tree property*: for any vertex x,

- for any vertex y in the left subtree of x, key[y] $\leq$ key[x],
- for any vertex z in the right subtree of x, key[z] $\geq$ key[x].

Definition 7.26 (AVL-tree)**.** An AVL-tree is a binary search tree that also satisfies the **AVL-property**: for any vertex x, |height[right[x]] - height[left[x]]| ≤ 1. In words, the height of its left and right subtrees differs by at most 1.

Theorem 7.27 (Number of vertices and height of an AVL-tree)**.** Denote the minimum number of vertices in a k-level ($k \geq 1$) AVL-tree by G_k. Then $G_k = F_{k+2} - 1$, where F_k is the kth Fibonacci number (see Definition 3.7). Thus the height of an AVL-tree is $O(\log n)$.

7.5 Shortest paths

Definition 7.28 (Predecessor subgraph)**.** After the run of the BFS, we define the following so-called *predecessor subgraph* G_π:

- vertices: the source and those who have a parent, formally $V_\pi = \{u : \pi[u] \neq \text{NIL}\} \cup \{s\}$,
- edges: edges connecting vertices to their parents, formally $E_\pi = \{(\pi[u], u), u \in V_\pi\}$.

Definition 7.29 (Breadth-first tree)**.** We call the predecessor subgraph a *breadth-first tree* if V_π is the out-component of s and for all u $\in V_\pi$, there is a unique $s \rightsquigarrow u$ path within G_π.

8 Algorithms

Note: the following algorithms may appear in the midterm. You do not have to write them exactly like this, e.g. you can change notation, and I can understand them in C or Python code too; but preserve the logic and the various cases/nuances.

List of Algorithms

Algorithm 1 Extended Euclidean algorithm (recursive version)

1: **EXT_EUCL_REC**(a,b,@d,@x,@y)
2: // INPUT: $a, b \in \mathbb{N}$, $a \geq b$
3: // OUTPUT: $d = \gcd(a, b)$, $x, y \in \mathbb{Z}$ coefficients of linear combination $d = x \cdot a + y \cdot b$
4: IF $b > 0$ THEN
5: $q \leftarrow \lfloor \frac{a}{b} \rfloor$
6: $r \leftarrow a - b \cdot q$
7: (d,x_old,y_old) ← EXT_EUCL_REC(b,r,@d,@x,@y)
8: x ← y_old
9: y ← x_old - y_old · q
10: ELSE
11: d ← a
12: x ← 1
13: y ← 0
14: **RETURN**(d,x,y)

Algorithm 2 Solver for linear congruence equation

1: **SOLVER_LCE**(a,b,n,@X)
2: // INPUT: $a, b, n \in \mathbb{Z}$ constants of the linear congruence equation
3: // OUTPUT: X, sequence of solutions (possibly empty with length[X] = 0, if no solutions exist), *indexed from 0*
4: (d,y,x) ← EXT_EUCL_REC(n,a,@d,@y,@x) // Write $d = xa + yn$.
5: IF $d|b$ THEN
6: length[X] ← d
7: $X_0 \leftarrow x\frac{b}{d} \mod n$
8: FOR i ← 1 TO d-1 DO
9: $X_i \leftarrow x_0 + i\frac{n}{d} \mod n$
10: ELSE
11: length[X] ← 0
12: **RETURN**(X)

Algorithm 3 Modular exponentiation

1: **MOD_EXP**(a,b,n,@c)
2: // INPUT: $a, b, n \in \mathbb{Z}$ // Suppose b is given in binary: $b = b_k b_{k-1} \dots b_1 b_{0(2)}$.
3: // OUTPUT: $c = (a^b \mod n) \in \mathbb{Z}$
4: $c \leftarrow 1$
5: FOR i ← k DOWNTO 0 DO
6: $c \leftarrow c^2 \mod n$
7: IF $b_i = 1$ THEN
8: $c \leftarrow c \cdot a \mod n$
9: **RETURN**(c)

Algorithm 4 Generation of RSA keys

1: **GEN_RSA_KEYS**(p,q,e,@P,@S)
2: // INPUT: p,q primes, $e \geq 3$ small odd integer
3: // OUTPUT: P private key and S secret key, if they exist
4: $n \leftarrow pq$
5: $f \leftarrow (p-1)(q-1)$
6: IF $\gcd(e, f) \neq 1$ THEN
7: RETURN("Keys don't exist.") // Or P ← NIL, S ← NIL, RETURN(P,S)
8: $d \leftarrow e^{-1} \mod f$ // $X \leftarrow$ SOLVER_LCE(e,1,f,@X), $d \leftarrow X_0$
9: P ← (e,n)
10: S ← (d,n)
11: **RETURN**(P,S)

Algorithm 5 Linear search in array

```
1: LIN_SEARCH(A,k,@x)
2: // INPUT: A array, k key to be found
3: // OUTPUT: x: index of a record that key[A_x] = k, or 0 if no such record exists
4: IF length[A] = 0 THEN
5:     x ← 0
6: ELSE
7:     x ← head[A]
8:     WHILE x ≤ end[A] AND key[A_x] ≠ k DO
9:         INC(x)
10:    IF x > end[A] THEN
11:        x ← 0
12: RETURN(x)
```

Algorithm 6 Binary search in *ordered* array, iterative version

```
1: BIN_SEARCH_IT(A,k,@x)
2: // INPUT: array A, key k to search
3: // OUTPUT: x index so that key[A_x]=k or 0 if no such record is found
4: IF length[A] = 0 THEN
5:     x ← 0
6:     RETURN(x)
7: a ← head[A]
8: b ← end[A]
9: WHILE a ≤ b DO
10:    c ← ⌊(a+b)/2⌋        // calculate midpoint
11:    IF key[A_c] = k THEN
12:        x ← c
13:        RETURN(x)
14:    ELSE IF k < key[A_c] THEN
15:        b ← c − 1
16:    ELSE
17:        a ← c + 1
18: x ← 0
19: RETURN(x)
```

Algorithm 7 Delete from doubly linked list

```
1: LL_DEL(L,x)
2: // INPUT: L linked list, x (non-NIL) pointer of an element to delete
3: IF x ≠ NIL THEN
4:     y ← prev[x]
5:     z ← next[x]
6:     IF y = NIL THEN
7:         head[L] ← z
8:     ELSE
9:         next[y] ← z
10:    IF z = NIL THEN
11:        end[L] ← y
12:    ELSE
13:        prev[z] ← y
14: RETURN()
```

Algorithm 8 Operations (push and pop) in queue, with linked list implementation

```
1: QUEUE_PUSH_LL(Q,x)
2: // INPUT: Q singly linked list for storing the queue, x pointer of new element
3: y ← end[Q]
4: IF y ≠ NIL THEN
5:     next[y] ← x
6: next[x] ← NIL
7: end[Q] ← x
8: RETURN()

9: QUEUE_POP_LL(Q,@x)
10: // INPUT: Q singly linked list for storing the queue
11: // OUTPUT: x pointer of the first element, or NIL if the queue is empty
12: x ← head[Q]
13: IF x ≠ NIL THEN
14:     head[Q] ← next[x]
15: RETURN(x)
```

Algorithm 9 Search in open address hash table

```
1: SEARCH(T,N,k,@i)
2: // INPUT: T hash table, N number of rows, k key
3: // OUTPUT: i index so that key[T_i] = k or NIL
4: t ← 0
5: i ← h_0(k)
6: s ← 1 + (k mod (N − 1))      // h_1 function, stepsize – for double hash
7: WHILE t ≤ N − 1 AND (status[T_i] = D OR (status[T_i] = O AND key[T_i] ≠ k)) DO
8:     INC(t)
9:     i ← h(k, t)      // general form, to be replaced
10:    i ← i + c        // linear trial, with general stepsize c
11:    i ← i + t        // quadratic trial, specific case
12:    i ← i + s        // double hash
13: IF t = N or status[T_i] = F THEN      // key k was not found
14:    i ← NIL
15: RETURN(i)
```

Algorithm 10 Partition algorithm (subroutine for Selection and Quicksort)

```
1: PARTITION(@A,a,b,x,@q)
2: // INPUT: A: array, a and b: first and last index of the interval to partition, x: the pivot, an
   element of A, within the index interval [a, b].
3: // OUTPUT: A array partitioned and q index, such that A_a, ..., A_q ≤ x, A_{q+1}, ..., A_b ≥ x.
4: i ← a − 1
5: j ← b + 1
6: WHILE TRUE DO      // Alternatively: i < j.
7:     REPEAT
8:         INC(i)
9:     UNTIL A_i ≥ x
10:    REPEAT
11:        DEC(j)
12:    UNTIL A_j ≤ x
13:    IF i < j THEN
14:        swap A_i ↔ A_j
15:    ELSE
16:        q ← j
17:        RETURN(A,q)
18: RETURN()      // In fact superfluous.
```

Algorithm 11 The merge algorithm (subroutine for Merge sort)

```
1: MERGE(A,B,@C)
2: // INPUT: sorted arrays A and B
3: // OUTPUT: sorted union C, length[C] = length[A]+length[B]
4: length[C] ← length[A]+length[B]
5: i ← 1      // Index in A.
6: j ← 1      // Index in B.
7: k ← 1      // Index in C.
8: WHILE i ≤ length[A] AND j ≤ length[B] DO
9:     IF A_i ≤ B_j THEN
10:        C_k ← A_i
11:        INC(i)
12:        INC(k)
13:    ELSE
14:        C_k ← B_j
15:        INC(j)
16:        INC(k)
17: WHILE i ≤ length[A] DO
18:    C_k ← A_i
19:    INC(i)
20:    INC(k)
21: WHILE j ≤ length[B] DO
22:    C_k ← B_j
23:    INC(j)
24:    INC(k)
25: RETURN(C)
```

Algorithm 12 Counting sort (binsort)

```
1: BINSORT(A,m)
2: // INPUT: array A to sort, m largest possible key value      // Denote n = length[A].
3: // OUTPUT: B sorted array      // length[B] = n
       // Stage 1: count values.
4: C ← [0, ..., 0]  (m times)
5: FOR i ← 1 TO n DO
6:     INC(C(A_i))      // If A_i = k, increase C_k.
       // Stage 2: cumulative counts.
7: FOR k ← 2 TO m DO
8:     C_k ← C_k + C_{k−1}
       // Stage 3: build ordered array in B.
9: length[B] ← length[A]
10: FOR i ← n DOWNTO 1 DO
11:    B(C(A_i)) ← A_i
12:    DEC(C(A_i))
13: RETURN(B)
```

Algorithm 13 The 3 tree traversal algorithms

```
1: INORDER_TRAVERSAL(x)
2: // INPUT: x: pointer to a vertex (possibly NIL)
3: IF x ≠ NIL THEN
4:     INORDER_TRAVERSAL(left[x])
5:     Print(key[x])
6:     INORDER_TRAVERSAL(right[x])
7: RETURN()

8: PREORDER_TRAVERSAL(x)
9: // INPUT: x: pointer to a vertex (possibly NIL)
10: IF x ≠ NIL THEN
11:     Print(key[x])
12:     PREORDER_TRAVERSAL(left[x])
13:     PREORDER_TRAVERSAL(right[x])
14: RETURN()

15: POSTORDER_TRAVERSAL(x)
16: // INPUT: x: pointer to a vertex (possibly NIL)
17: IF x ≠ NIL THEN
18:     POSTORDER_TRAVERSAL(left[x])
19:     POSTORDER_TRAVERSAL(right[x])
20:     Print(key[x])
21: RETURN()
```

Algorithm 14 Insert into heap (max-heap)

```
1: HEAP_INSERT(@A,x,@message)
2: // INPUT: A array with heap property, x new element
3: // OUTPUT: A updated heap, message
4: IF heapsize[A] = arraysize[A] THEN
5:     message ← "heap is full"
6:     RETURN(A,message)
7: INC(heapsize[A])
8: c ← heapsize[A]
9: A_c ← x
10: p ← ⌊c/2⌋
11: WHILE p > 0 AND A_p < A_c DO
12:     swap A_c ↔ A_p
13:     c ← p
14:     p ← ⌊c/2⌋
15: message ← "successfully inserted"
16: RETURN(A,message)
```

Algorithm 15 Search in binary search tree (iterative version)

```
1: BST_SEARCH(T,k)
2: // INPUT: T: binary search tree T, k key
3: // OUTPUT: pointer x with key[x] = k, or NIL
4: x ← root[T]
5: WHILE x ≠ NIL DO
6:     IF key[x] = k THEN
7:         RETURN(x)
8:     ELSE IF k < key[x] THEN
9:         x ← left[x]
10:    ELSE      // k > key[x] case
11:        x ← right[x]
12: RETURN(x)
```

Algorithm 16 Breadth-first search on G from source vertex s

```
1: BREADTH_FIRST_SEARCH(G,s,@dist,@π)
2: // INPUT: unweighted graph G, source vertex s
3: // OUTPUT: dist, π arrays indexed by vertices, d: distance from s, π: predecessor on a shortest
   path
4: FOR EACH u ∈ G \ {s} DO
5:     color[u] ← white
6:     dist[u] ← ∞
7:     π[u] ← NIL
8: color[s] ← grey
9: dist[s] ← 0
10: π[s] ← NIL
11: Q ← {}        // initialize empty queue
12: PUSH(@Q,s)
13: WHILE Q ≠ {} DO
14:     u ← POP(@Q)
15:     FOR EACH v ∈ (out-)neighbors[u] DO
16:         IF color[v] = white THEN
17:             color[v] ← grey
18:             dist[v] ← dist[u]+1
19:             π[v] ← u
20:             PUSH(@Q,v)
21:     color[u] ← black
22: RETURN(dist,π)
```

References

[1] Attila Házy. *Adatstruktúrák és algoritmusok, Definíciók és tételek 1.* https://www.uni-miskolc.hu/~matha/adatstr_definiciok_tetelek.pdf.

[2] Attila Házy. *Adatstruktúrák és algoritmusok, Definíciók és tételek 2.* https://www.uni-miskolc.hu/~matha/adatstr_definiciok_tetelek_2zh.pdf.