Data structures and algorithms
Course lecture notes

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

Latest version: March 11, 2025

Contents

Part I

Introduction and representation of numbers

Chapter 1

Lecture

Back to top

Contents of chapter

1.1 Introduction, motivation

To demonstrate the aim of this course, consider the example of developing navigation software. From the conception to realization, development of this software goes through different stages and levels:

1. Specification – the user specifies desired functionality.

   In our example, we might want to be able to find the shortest route, or the fastest one. For car navigation, we may wish to exclude highways with toll; when using public transport, we may wish to limit the number of connections, or set bounds on connection times.

2. Modeling and planning

   a. **The model.** A *model* is a simplification of reality that only focuses on relevant details. For the same real-world problem, various models with different amounts of detail can be built. In our example of navigation, a road being one-way is vital, however it being a dirt road might or might not be relevant. The model determines what data we must store and analyze.
   b. **The algorithm.** Given the data, we must determine a way for the computer to process it and come to a solution – we must choose an *algorithm*. We think of an algorithm as a

step-by-step instruction that takes some data, that is, some parameters of the model, as its input, and returns a solution[1] as its output.

   c. **The data structures.** The same data can be stored in different ways: a teacher may have a notebook, an Excel sheet, or use the Neptun system to manage grades. It matters wether the list of students is alphabetically ordered, etc. There is always a trade-off, different ways of storing the same data will make certain operations faster, while others slower. The choice of the data structures goes hand-in-hand with the choice of the algorithm: we want data structures that suit and support the algorithm and make it run as fast as possible.

3. Coding – beyond the planning stage, a programming language is chosen and the algorithm and data structures are implemented.

In this course, we mainly concern ourselves with the modeling and planning stage of development. We will study data structures and algorithms, certainly. We will also consider why it's crucial to choose the right algorithm and the right data structures that help it run most optimally, to optimize the runtime of the final program. All of this is part of the discipline of computer science – it is computer oriented, but relies on mathematics quite a lot.

1.2 Introduction to programming and algorithms

1.2.1 What is a computer?

It's just a computing machine. The computer isn't as smart as you'd think, looking at your smartphone or ChatGPT. It looks smart because smart people have written layers and layers of more and more complex programs that run on top of each other.

Why might we want to do programming? Maybe, we're trying to solve a complex logistics problem, which has too much data to do by hand, and there's no pre-existing program that can help (or there's licencing or data breach issues and we choose not to use any).

When we want to program, we have to think of the computer as but a silly machine that cannot think for itself and can only follow our instructions precisely. Such instructions include reading and writing variables, comparing two variables, jumping somewhere in the code based on the result of a conditional, or doing some (mathematical) operations on one or more variables.

1.2.2 Program, algorithm

Definition 1.1 (Program)**.** A sequence of instructions for the computer to execute.

Definition 1.2 (Procedure)**.** A unit of a program for a specified task. It communicates with the rest of the program through its input and output variables.

Definition 1.3 (Algorithm)**.** A specified, step-by-step calculation method, a tool for solving computation problems.

We can think of an algorithm as existing on the mathematical, modeling plane, but we may also refer to a program or procedure implementing this algorithm as "the algorithm".

For the same problem, we may ome up with various mathods and approaches to solve it, which lead to different algorithms. However, some of them may be smarter, faster and more efficient. This efficiency becomes very important when we're working with large data, if we don't want to sit around and wait days or even centuries for a solution. We'll talk more later about how we measure the speed of an algorithm, and demonstrate that it's indeed possible for an inefficient algorithm to run for centuries even for small or reasonable amounts of data.

[1] Possibly not the best solution, or possibly multiple solutions. Some problems are too hard and we can only calculate approximate solutions within a reasonable timeframe. We'll get back to this.

1.2.3 Common principles for planning algorithms

Of course, these are not the end-all-be-all. Creativity always has its place. But these principles can serve as building blocks or insporation and often help us to get started in designing an algorithm for a new problem.

Many of them are based on structured repetition. Unlike humans, computers do not get tired of repetition. And using repetition the right way can also save us from writing long instructions.

Some of these ideas may also overlap, an algorithm may be classified as several at once.

- creating *procedures* can be thought of as a form of repetition. It is a unit of code that can be called again and again with different inputs, whenever suitable.
- *loops* are a local form of repetition, a chunk of code that can be executed a given number of times, or while certain conditions are met.
- *iteration* refers to repetition that constructs a solution from the ground up – a bottom-up approach.
- *recursion* reduces the problem into smaller subproblems that can be solved the same way – a top-down approach. This is closest to the analogy of task delegation. For a reucrsive solution to work in finite time, we must be cautious. We have to make sure the simplest cases can be handled, and each time we delegate, we must *actually* reduce the problem, to guarantee that we reach the simplest case in a finite number of steps.
- *divide and conquer* is a form of recursion where a problem is split into independent subproblems that can be solved separately, and then the partial solutions can be combined.
- *dynamic programming* is an iterative approach. We construct a sequence of subproblems that are nested like Matryoshka dolls (Russian nesting dolls), that can be solved sequentially (based on the previous partial results) in the same way.

1.3 Introduction to data

1.3.1 What is data?

Definition 1.4 (Data)**.** A (relevant) piece of information or detail used to qualify or quantify someone or something.

We think of data as existing in the real world.

Definition 1.5 (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 1.6 (Abstract data type)**.** Abstract data type consists of the feasible set of abstract data and the set of operations defined on this set.

Abstract data and abstract data type live on a theoretical plane, in the world of mathematics, when we make a mathematical model of the world, without yet implementing it on the computer.

Here, we already have an idea of how to represent reality – for example, we may encode cities by zip code (postal code). What operations are allowed on the data is an important part of this representation – for example, it makes no sense to add or multiply zip codes, even if they are numbers. But it makes sense to check if it falls within a certain interval of zip codes that signifies a larger geographic region.

Definition 1.7 (Data structure)**.** A specific, complete implementation of an abstract data type, including the implementation of the data as well operations on it.

Data structure is the already implemented (aka coded) version, and lives in the computer.

1.3.2 Storing data

The computer's memory consists of a linear sequence of bits that have two possible states, that we read as 0 and 1.

The smallest unit we can read and write from memory is one *byte*, which is a string of 8 bits. Sometimes we only use parts of this, but most often the whole 8-bit sequence is read together. Reading this sequence from left to right, we index the bits by 7 to 0. The leftmost is referred to as MSB = most significant bit, and the rightmost is referred to as LSB = least significant bit. This 8-bit sequence can take $2^8 = 256$ different values.

Sometimes that is too few possibilities, then we can join several bytes together for larger storage units. Two bytes joined (16 bits) is called a *word*, and two words joined (32 bits) is called a *double word*. On many systems, we read them left to right and index their bits from 15 to 0 and 31 to 0, respectively.

Let us also emphasize that there is no such thing as an "empty bit", there are always 0s and 1s. The distinction is whether we put that data there, or it's a leftover from a previous program. That is why we must be careful, since we can always read and interpret the bits that are there. The question is whether we should, if it is meaningful to us, or memory junk.

Memory and file sizes are measured in the following units:

- bit, b
- byte, B = 8 bits
- kilobtye, kB = 1024 B
- megabtye, MB = 1024 kB
- gigabtye, GB = 1024 MB
- terabtye, TB = 1024 GB

Note: usually, the prefix kilo- means a 1000 times. However, in computer science we often think in terms of powers of 2, and $2^{10} = 1024$ is close enough to 1000 to borrow the same prefix.

1.4 Character and logical data types

In general, data types we use include logical type (true or false), characters (that we can use to build text documents) and numbers. Numbers will be discussed in detail in the instructions part, we briefly touch on the character and logical types here. We discuss them mostly as abstract data types, with some notes on implementation.

1.4.1 Characters

The concept of *characters* includes everything we may type, in any writing system – from letters of the Latin alphabet, digits and punctuation marks (comma, period, etc.), through whitespace characters (tabulator, newline, etc.), Cyrillic and Greek alphabets, to complex characters such as Chinese characters and even emojis.

These are stored as so-called *character tables* – a map from bit sequences to the characters they denote. We can also think of them as maps from numbers, called the *code* of the character, to letters.

The earliest was the ASCII table – it included the Latin alphabet, digits, punctuation, whitespace and some command characters (such as for the computer to make a beep). The codes ranged from 0 to 127, and they could be stored on 7 bits. For convenience, one character was stored per byte unit. This allowed the ASCII table to be extended, still only using 1 byte per character, and for European languages, various local code tables were introduced to include the local alphabets or accented letters.

However, documents were not portable under this system, and did not solve the issue for more complex Asian writing systems.

These issues were resolved in Unicode, that assigns a unique code to each characters (no more competing character tables!), and allows for several million code points, including all alphabets and writing systems around the world, and even emojis. It includes the original ASCII characters on their original 0 to 127 codes. Unicode has several implementations, so-called encodings, UTF-8, UTF-16 and UTF-32. UTF-32 is the easiest one as it stores each character on 32 bits, however it is quite wasteful for Western texts when ASCII characters could fit in 1 byte only. UTF-8 is most commonly used over the internet, it is more efficient in storage as it uses a varying encoding length, a character can take from 1 to 4 bytes, and the least necessary number of bytes is used. We do not go into details on the encoding, but it uses quite smart bit patterns to identify the role of each byte, whether it stands alone, starts or continues a multi-byte unit encoding a single character.

To go from a single character to a text, we make a string: a sequence of characters. We only need to know the character table to use and how long the sequence is. For the latter, there are different conventions,we may specify its length in advance, or terminate it with a special character. We won't need more detail on this.

1.4.2 Logical type

The logical type takes values in the set $L = \{\text{T,F}\}$, or $\{\text{TRUE, FALSE}\}$, or $\{1, 0\}$. In computer science, it arises when evaluating conditions. Since it can only take two possible values, it can be stored on a single bit.

To create complex conditions, 3 basic operations are enough: unary operation of negation (NOT), and binary operations conjunction (AND) and disjunction (OR). They are quite inuitive: NOT x is true when x is false; x AND y is true when both are true; x OR y is true when at least one is true. Formally, we define them in *operation tables*: defining the output of the operation function for each possible input.

Unary operation

	NOT
x	$\bar{x}$
0	1
1	0

Binary operations

		AND	OR
x	y	$x \wedge y$	$x \vee y$
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	1

Using these operation tables, we can come up with more operations – as many different ways as there are to fill the function column with 0s and 1s. There are names for each possible non-trivial operation, but we do not need to go deeper into them. That is because all of them can be expressed as a combination of NOT, AND and OR.[2] In programming practice most of the conditions we need naturally appear as a combination of NOT, AND and OR.

[2]For example, a DNF (disjunctive normal form) can be used to transform any operation into a compbination of these three basic operations.

Chapter 2

Instruction

Back to top

Contents of chapter

2.1 Some mathematics functions

Definition 2.1 (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 the *closest* integer that isn't greater than x. Formally:

$$\lfloor x \rfloor = \max\{k \in \mathbb{Z} \mid k \leq x\}, \tag{2.1}$$

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 2.2 (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 the *closest* integer that isn't smaller than x. Formally:

$$\lceil x \rceil = \min\{k \in \mathbb{Z} \mid k \geq x\}, \tag{2.2}$$

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

Definition 2.3 (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:

$$\text{Round}(x) = \left\lfloor x + \frac{1}{2} \right\rfloor \tag{2.3}$$

Definition 2.4 (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,

$$\{x\} = x - \lfloor x \rfloor. \tag{2.4}$$

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

Example 2.1 (Floor, ceil, round and fraction functions).

x	-6	-5.8	-5.5	-5.2	-5	5	5.2	5.5	5.8	6
$\lfloor x \rfloor$	-6	-6	-6	-6	-5	5	5	5	5	6
$\lceil x \rceil$	-6	-5	-5	-5	-5	5	6	6	6	6
Round(x)	-6	-6	-5	-5	-5	5	5	6	6	6
$\{x\}$	0	0.2	0.5	0.8	0	0	0.2	0.5	0.8	0

Definition 2.5 (Whole quotient, div operation)**.** Let a and b be integers. We define the whole quotient (result of whole division) as

$$a \text{ div } b := \begin{cases} \left\lfloor \dfrac{a}{b} \right\rfloor & \text{if } b \neq 0, \\ \text{not defined} & \text{if } b = 0. \end{cases} \tag{2.5}$$

Definition 2.6 (Whole remainder, mod operation)**.** Let a and b be integers. We define the whole remainder as

$$a \text{ mod } b := \begin{cases} a - (a \text{ div } b) \cdot b = a - \left\lfloor \dfrac{a}{b} \right\rfloor \cdot b, & \text{if } b \neq 0, \\ a & \text{if } b = 0. \end{cases} \tag{2.6}$$

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

Exercise 2.1 (div and mod operations)**.** Calculate $\pm 12 \text{ div } \pm 5$ and $\pm 12 \text{ mod } \pm 5$, for all possible combinations of signs.

2.2 Number systems

This is some mathematics we introduce that will be necessary for understanding how we store numbers in the computer.

2.2.1 Base ten

The usual way we write numbers is in the decimal system. When we write 2024, we mean *two thousand twenty four*, or when we write 2.5, we mean *two and one half.* In this number system, and the meaning of the same digit changes based on its location or *place value*. In 2024, the first 2 stands in the place value of *thousands*, and thus means two thousand. The second 2 stands in the place value of *tens*, and thus means twenty. The place values are the powers of our base 10 (*ten*). Formally, we can write

$$2024 = 2 \cdot 10^3 + 0 \cdot 10^2 + 2 \cdot 10^1 + 4 \cdot 10^0. \tag{2.7}$$

For fractions, such as 2.5, we can write

$$2.5 = 2 \cdot 10^0 + 5 \cdot 10^{-1}. \tag{2.8}$$

After the decimal dot, negative powers of the base appear. Power 0, the place value of *ones*, appears right before the decimal dot.

We also note that digits in base 10 (*ten*) range from 0 to 9, *ten* itself is written as 10, for 1 of *tens* and 0 of *ones*.

2.2.2 General base

We prefer base *ten*, since we have *ten* fingers to count on. But we can make sense of a similar representation of numbers with any base $b \in \mathbb{Z}$ such that $b \geq 2$, where the place values or powers of b instead. If b is at most ten, our usual digits are enough. For b larger than ten, for the sake of readability, we use capital letters to represent digits, $A = 11$, $B = 12$, …, $Z = 35$. That is enough up until base thirty-six, and here we are not interested in based higher than that.

We write and interpret base b numbers as follows:

$$\begin{aligned} c_n c_{n-1} \dots c_1 c_{0\,(b)} &= c_n \cdot b^n + c_{n-1} \cdot b^{n-1} + \dots + c_1 \cdot b^1 + c_0 \cdot b^0 \\ &= c_n \cdot b^n + c_{n-1} \cdot b^{n-1} + \dots + c_1 \cdot b^1 + c_0 \end{aligned} \tag{2.9}$$

and for fractions,

$$0\,.\,c_{-1} c_{-2} \dots c_{-m\,(b)} = 0 + c_{-1} \cdot b^{-1} + c_{-2} \cdot b^{-2} + \dots + c_{-m} \cdot b^{-m}. \tag{2.10}$$

Here, all digits c_i are integers and $0 \leq c_i < b$. We always denote the base in lower index (b) in base ten, as b in base b will be written as 10 – read "one zero".

If no base is written, we assume base ten.

We read out numbers in base b digit by digit, to avoid confusion. For example, we read $102_{(3)}$ as "one-zero-two in base three", which equals $1 \cdot 9 + 2 \cdot 1 = 11$ eleven.

2.2.2.1 Number of digits

The following theorem shows how many digits a number takes in a given base, without having to do the conversion to that base.

Theorem 2.1 (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

$$x = c_n c_{n-1} \dots c_1 c_{0\,(b)}. \tag{2.11}$$

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

Proof. If b^n is the largest place value with a nonzero c_n digit, that means

$$b^n \leq x < b^{n+1}, \tag{2.12}$$

that is,

$$n \leq \log_b x < n + 1. \tag{2.13}$$

Hence by definition $n = \lfloor x \rfloor$. The number of digits is $n + 1$, since we also have c_0 in the place value of ones. □

2.2.2.2 Trade-off between bases

We prefer base *ten*, or the decimal system, not only because we have ten fingers, but also because it is a number that is neither two small nor too large.

If the base is too large, we must introduce many different digits. If the base is too small, even small numbers must be written with many digits, see the theorem above. Both of these can cause confusion and readability issues around numbers.

Historically and across cultures, numbers have been written differently. For example, Roman numerals have their own system, that isn't founded on bases, place values and digits, where year numbers may look pretty, but even doing addition becomes a nightmare. Even when and where a number system was used, it was not always base ten. For example, base twelve was popular, as it has the advantage that twelve can be divided equally into 2, 3 and 4 parts. Base sixty has the advantage that it can be divided equally into 2, 3, 4 and 5 parts as well, but it is a base way too large to invent enough digits. The way we measure time today, with two times twelve hours in a day and sixty minutes in an hour, is a relic of different number systems used in ancient times (by Babylonians, specifically). The names of numbers in English, as well the usage of dozens, also bare marks of base twelve. The numbers in French language bare marks of various number systems.

2.2.2.3 Preferred bases in computer science

To represent numbers on the computer, the storage units of *bits* can take two states. We can interpret these as 0 and 1, which lends itself well to a base 2 number system.

However, base two becomes unreadable with long sequences consisting only of 0s and 1s. Hence we often convert those numbers to hexadecimal, that is, to base sixteen.

2.2.3 Conversion between number systems

We will most often need conversion between some base b and our preferred base ten, and in computer science, also conversion between base two and base sixteen. (The methods below work more generally, if we substitute a different base b' for ten.)

From base b to base ten, we can always convert by definition, using the place values of powers of b. However, there is a faster and more efficient way, called the Horner method.

Statement 2.1 (Digit shift)**.** We know that in base ten, multiplication (respectively, division) by ten shifts the digits left (resp., right). In base b, multiplication (resp., division) by b does the same:

$$c_n \dots c_1 c_{0\,(b)} \cdot b = c_n \dots c_1 c_0 0_{(b)} \tag{2.14}$$

2.2.3.1 Horner method for integers

Based on this observation, we can convert a base b number to base ten left-to-right:

1. take c_n
2. multiply by b and add c_{n-1}
3. repeat: multiply previous result by b and add next digit.

4. repeat until you add c_0.

This works, since

$$c_n \dots c_{k\,(b)} \cdot b + c_{k-1} = c_n \dots c_k 0_{(b)} + c_{k-1} = c_n \dots c_k c_{k-1\,(b)}. \tag{2.15}$$

We can also think of this method as rearranging the definition of a base b number as follows:

$$\begin{aligned} &c_n \cdot b^n + c_{n-1} \cdot b^{n-1} + c_{n-2} \cdot b^{n-2} + \dots + c_1 \cdot b + c_0 \\ &= \Bigg(\Bigg(\cdots \Big((c_n \cdot b + c_{n-1}) \cdot b + c_{n-2} \Big) \cdots \Bigg) \cdot b + c_1 \Bigg) \cdot b + c_0 \end{aligned}$$

If we expand the brackets, each c_i is is indeed multiplied by b a total of i times.

Example 2.2. For computations by hand, it is handy to write this as a table. We show both the general computation and the conversion of $\overrightarrow{11011010}_{(2)}$.

	b
c_n	c_n
$\vdots$	$\vdots$
c_k	x
c_{k-1}	$x \cdot b + c_{k-1}$
$\vdots$	$\vdots$
c_1	y
$\downarrow c_0$	$y \cdot b + c_0$

	2
1	1
1	$1 \cdot 2 + 1 = 3$
0	$3 \cdot 2 + 0 = 6$
1	$6 \cdot 2 + 1 = 13$
1	$13 \cdot 2 + 1 = 27$
0	$27 \cdot 2 + 0 = 54$
1	$54 \cdot 2 + 1 = 109$
$\downarrow$ 0	$109 \cdot 2 + 0 = 218$

2.2.3.2 Horner method for fractions

Analogously to above, we can convert a base b fraction to base ten right-to-left:

1. take c_{-n} and divide by b
2. add c_{-n+1} and divide by b
3. repeat: take previous result, add next digit, and divide by b.
4. in the last step, you add c_{-1} and divide by b.

There is a slight difference from the integer version, because the place value of ones, that is b^0, is only present in the integer part.

This works, since

$$(0\,.\,c_{-k} \dots c_{-m\,(b)} + c_{-k+1})/b = c_{-k+1}\,.\,c_{-k} \dots c_{-m\,(b)}/b = 0\,.\,c_{-k+1} c_{-k} \dots c_{-m\,(b)} \tag{2.16}$$

We can also think of this method as rearranging the definition of a base b fraction as follows:

$$\begin{aligned} &0 + c_{-1} \cdot b^{-1} + c_{-2} \cdot b^{-2} + \dots + c_{-m+1} \cdot b^{-m+1} + c_{-m} \cdot b^{-m} \\ &= \Bigg(\Bigg(\cdots \Big((c_{-m}/b + c_{-m+1})/b \Big) \cdots \Bigg)/b + c_{-1} \Bigg)/b \end{aligned}$$

If we expand the brackets, each c_{-i} is divided by b a total of i times. This is why we now move right-to-left. We always start furthest from the fraction point and moving towards it, as the digits furthest away have to be multiplied or divided by the base the most times.

Example 2.3. For computations by hand, it is again handy to write this as a table. We show both the general computation and the conversion of $0.\overleftarrow{1101}_{(2)}$.

	b
c_{-m}	c_{-m}/b
$\vdots$	$\vdots$
c_{-k}	x
c_{-k+1}	$(x + c_{k-1})/b$
$\vdots$	$\vdots$
c_1	y
c_0	$(y + c_0)/b$

	2
1	$1/2 = 0.5$
0	$(0.5 + 0)/2 = 0.25$
1	$(0.25 + 1)/2 = 0.625$
1	$(0.625 + 1)/2 = 0.8125$

2.2.3.3 Conversion to base b – integers

Similar to Statement 2.1,

$$c_n \dots c_1 c_{0\,(b)} / b = c_n \dots c_1 \,.\, c_{0\,(b)}, \tag{2.17}$$

or in other words, the whole division (Euclidean division) by b results in

$$c_n \dots c_1 c_{0\,(b)} = c_n \dots c_{1\,(b)} \cdot b + c_0. \tag{2.18}$$

That is, we are able to separate the *last* digit and the rest of the number with the whole division; in other words, we can obtain them separately with the div and mod operators.

The procedure for obtaining a number's digits in base b, if its value or decimal form is known:

1. execute whole division by b on the number and note down the remainder.
2. repeat with the whole quotient, execute whole division again, always writing the new digit *left* of the previous digits.
3. stop when whole quotient is 0.

Example 2.4. For computations by hand, it is again handy to write this as a table. We show both the general computation and the conversion of 87 to base two.

x	$\div b$
$x_1 := x \text{ div } b$	$x \text{ mod } b =: c_0$
$x_2 := x_1 \text{ div } b$	$x_1 \text{ mod } b =: c_1$
$\vdots$	$\vdots$
x_{n-1}	?
$0 = x_n := x_{n-1} \text{ div } b$	$x_{n-1} \text{ mod } b =: c_n$

87	$\div 2$
$87 \text{ div } 2 = 43$	$87 \text{ mod } 2 = 1$
$43 \text{ div } 2 = 21$	$43 \text{ mod } 2 = 1$
$21 \text{ div } 2 = 10$	$21 \text{ mod } 2 = 1$
$10 \text{ div } 2 = 5$	$10 \text{ mod } 2 = 0$
$5 \text{ div } 2 = 2$	$5 \text{ mod } 2 = 1$
$2 \text{ div } 2 = 1$	$2 \text{ mod } 2 = 0$
$1 \text{ div } 2 = 0$	$1 \text{ mod } 2 = 1$

$$87 = \overrightarrow{1010111}_{(2)}.$$

Remember that we got the last digit first, hence we read the number from bottom to up.

2.2.3.4 Conversion to base b – fractions

Similar to Statement 2.1,

$$0 \,.\, c_{-1} c_{-2} \dots c_{-m\,(b)} \cdot b = c_{-1} \,.\, c_{-2} \dots c_{-m\,(b)}. \tag{2.19}$$

That is, by multiplying by the base b, we are able to separate the *first* digit after the fraction point as the integral part of the number.

The procedure for obtaining a fraction's digits in base b, if its value or decimal form is known:

1. multiply the number by b and note down the integral part as the first digit after the fraction point.
2. repeat with the fractional part, multiply by b again and note down the integral part, always writing the new digit *right* of the previous digits.
3. stop when either:
   - the fractional part is 0, then the fraction is finite.
   - the current fractional part has already appeared, then the fraction is infinite but periodic.
   - required precision or computation limit is reached, fraction might be infinite and aperiodic (or one of the above cases, but the length or period is too long).

Example 2.5. For computations by hand, it is again handy to write this as a table. We show both the general computation and the conversion of 0.6875 to base two.

$b\cdot$	x
$c_{-1} := \lfloor x \cdot b \rfloor$	$x_1 := \{x \cdot b\}$
$c_{-2} := \lfloor x_1 \cdot b \rfloor$	$x_2 := \{x_1 \cdot b\}$
$\vdots$	$\vdots$
$\downarrow$ $c_{-m} := \lfloor x_{-m+1} \cdot b \rfloor$	$x_{-m} := \{x_{-m+1} \cdot b\} = 0$

The finite fraction case.

$2\cdot$	0.6875
$\lfloor 0.6875 \cdot 2 \rfloor = 1$	$\{0.6875 \cdot 2\} = .375$
$\lfloor 0.375 \cdot 2 \rfloor = 0$	$\{0.375 \cdot 2\} = .75$
$\lfloor 0.75 \cdot 2 \rfloor = 1$	$\{0.75 \cdot 2\} = .5$
$\downarrow$ $\lfloor 0.5 \cdot 2 \rfloor = 1$	$\{0.5 \cdot 2\} = .0$

$$0.6875 = 0.\overrightarrow{1011}_{(2)}.$$

Remember that we got the digit closest to the fraction point first, hence we read the number from top to bottom.

In this case, I personally prefer to write the starting number and the other fractions below it all on the right hand side. This is because when I'm doing the multiplication on paper by hand, it is easier to write the integral part on the left and the fractional part on the right, and split the two across the table cells.

2.2.3.5 Conversion between base two and sixteen

We demonstrate conversion between base two and sixteen via an example. Our key observation will be that $2^4 = 16$. We try to discover 2^4 and its powers, $(2^4)^2 = 2^8$, $(2^4)^3 = 2^{12}$, etc., in the expansion of the base-two number.

$$\begin{aligned}
1011010100_{(2)} &= 1 \cdot 2^9 + 0 \cdot 2^8 + 1 \cdot 2^7 + 1 \cdot 2^6 + 0 \cdot 2^5 + 1 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 0 \\
&= 2^8 \cdot \left(1 \cdot 2^1 + 0\right) + 2^4 \cdot \left(1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1\right) \\
&\quad + \left(0 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 0\right) \\
&= 16^2 \cdot 2 + 16 \cdot 13 + 4 \\
&= 2D4_{(16)}
\end{aligned} \tag{2.20}$$

We note that $10_{(2)} = 2 = 2_{(16)}$, $1101_{(2)} = 13 = D_{(16)}$, and $100_{(2)} = 4 = 4_{(16)}$. We conclude that we can convert a base-two number to base sixteen simply by grouping the digits by fours and converting each group to a digit of base sixteen. Note that the grouping starts from the right, always from the fraction point.

Conversely, we can convert a base-sixteen number to base two by writing each digit as a base-two number *on exactly four digits*, with leading zeros if necessary.

A similar conversion works for fractions as well, however in this case digits must be grouped starting from the left, always from the fraction point.

Table 2.1: Digits of base sixteen converted to base ten and two

Base 16	Base 10	Base 2	Base 16	Base 10	Base 2
0	0	0000	8	8	1000
1	1	0001	9	9	1001
2	2	0010	A	10	1010
3	3	0011	B	11	1011
4	4	0100	C	12	1100
5	5	0101	D	13	1101
6	6	0110	E	14	1110
7	7	0111	F	15	1111

This works more generally between bases where one base is the power of the other. For example, between base two and base eight, since $8 = 2^3$, the digits of the base-two number can be grouped by threes and converted to base-eight digits.

Later in this semester, it will often useful if we can quickly recognize digits of base sixteen, as well as quickly convert them to and from base two (with exactly four digits). We compiled a conversion table in Table 2.1.

2.3 Integer data types

Here, we talk a bit about how *integers* are stored on the computer. The storage of real numbers as *floating point numbers* will get its own big section.

2.3.1 Unsigned integer

This is the easiest. We just take the sequence of bits in the byte (or word, or double word) as a sequence of 0s and 1s, and read it as a non-negative binary number.

Unsigned integer

storage	range
1 byte (8 bits)	0 – 255
word (16 bits)	$0 - 2^{16} - 1$
double word (32 bits)	$0 - 2^{32} - 1$

2.3.2 Signed integer

There are several approaches to include negative numbers, here we talk about the *complement of twos* here. As the first step, we once again read the entire bit sequence as a binary number. Then, we look at the MSB. If it is 0, the non-negative number we read is the signed number stored. If the MSB is 1, that signifies a negative number; however, to get negative numbers with smaller absolute value, that is, closer to 0, we do not simply change the sign. We instead subtract a power of 2, and the power is the number of bits we store the signed integer on. Suppose we store this unsigned integer on 1 byte, i.e. 8 bits, then we subtract $2^8 = 256$ (when we store in a word or double word, we subtract 2^{16} or 2^{32} instead). This way we get a negative number, but not too far away from 0. Below, we show two tables to demonstrate some values for signed integer on 1 byte, and the ranges on different storage units.

Some values of the signed integer on 1 byte

bit sequence	read as binary number	value as signed integer
00000000	0	0
01111111	127	127
10000000	128	128 -256 = -128
11111111	255	255 -256 = -1

Ranges of signed integer type

storage	shift when MSB is 1	range
1 byte (8 bits)	$2^8 = 256$	-128 – +127
word (16 bits)	2^{16}	-2^{15} – $+(2^{15}-1)$
double word (32 bits)	2^{32}	-2^{31} – $+(2^{31}-1)$

The advantage of the *complement of twos* method over say, dedicating a bit to store the sign separately, is that it makes addition easier. With a dedicated sign bit, we'd have to add conditionals, check whether the addition turn into subtraction, etc. With the complement of twos, we can simply add the two numbers, forget the overflow (possible extra bit appearing before the MSB), and we get the right number with the right sign. (For those of you more versed in algebra, you can also think of this resulting system of set of numbers and operations as $\mathbb{Z}_{256}$, the ring of integers modulo 256 – or replace 256 with the appropriate power of 2.)

2.4 Floating point numbers

Floating point number is a common implementation of real numbers for computer storage.

2.4.1 Format and encoding steps

For this, we first require the *normal form* (canonical form) of a number. This is a form that focuses on the first couple non-zero digits, that we call the most significant digits, and the order of magnitude of a number. Precisely, we move the fractional dot after the first non-zero digit and multiply with the appropriate power of the base. For example, the normal form of some numbers:

$$-9856279 = -9.856279 \cdot 10^6 \tag{2.21}$$

$$0.00000000000256 = 2.56 \cdot 10^{-12} \tag{2.22}$$

$$10110101_{(2)} = 1.0110101 \cdot 2^7 \tag{2.23}$$

$$-0.0000101_{(2)} = -1.01_{(2)} \cdot 2^{-5} \tag{2.24}$$

For computer storage, the *base two normal form* is most efficient. Base two, since the digits are 0s and 1s. And the normal form, since we focus on the significant digits and the order of magnitude, instead of storing lots of leading 0s, or unsignificant digits that may cause the most significant digits to overflow.

We note that in the base two normal form, the single digit before the fractional dot is always 1, as it cannot be 0.

We can now introduce the floating point number format. We focus on single precision mostly, and briefly mention double precision.

A single precision floating point number is stored on 4 bytes, that is, 32 bits, that are utilized as follows:

- the leftmost *1 bit* is assigned for the *sign* of the number – by convention, a 1 sign bit means a negative number, and a 0 sign bit means a positive number (sign bit s is interpreted as $(-1)^s \cdot \ldots$).

- the next 8 bits are assigned to the *exponent* in the base two normal form. To avoid the complications around signed integers, we *offset* the exponent by +127, and store the resulting shifted/offset exponent as an unsigned integer. However, values and 0 and 255 have a special meaning (explained later). The normal range of the offset exponent is from 1 to 254, and thus for the actual exponent, from -126 to +127. (This is why the shift is 127 instead of 128, so that we have more positive exponents allowed.)
- the remaining 23 bits store the so-called *mantissa* or *significant*. These are the significant digits. We only store the *fractional part* in the normal form, since the 1 before the fractional dot is evident, and by not storing it, we can get more precise with the fractional part instead.

The steps of encoding/storing a number, originally given in base ten, as a floating point number:

0. given a number in base ten
1. convert the number to base two (convert integer and fractional part separately)
2. create base two normal form
3. offset the exponent and convert the offset exponent to base two (as unsigned integer)
4. write the sequence of 32 bits: 1 sign bit + 8 bits of the offset exponent + 23 bits of significant (only fractional part)
5. for readability, convert to hexadecimal, i.e., base sixteen: by groups of 4 bits, convert into 8 digits in base sixteen.

For decoding, i.e., if the bytes are given and we want to know what number is stored as a floating point number, follow the same steps backwards.

2.4.2 Example of encoding

0. given -100.375
1. convert to base two (using the methods learnt in week 1, left to the reader): $-100.375 = -1100100.011_{(2)}$
2. base two normal form $-1100100.011_{(2)} = -1.100100011_{(2)} \cdot 2^{+6}$
3. offset exponent is $6 + 127 = 133 = 10000101_{(2)}$
4. the bit sequence: `1100 0010 1100 1000 1100 0 ...0`
5. in hexadecimal: `C2 C8 C0 00`

2.4.3 Example of decoding

5. in hexadecimal: `43 7A E0 00`
4. the bit sequence: `0100 0011 0111 1010 1110 0 ...0`
3. offset exponent is $10000110_{(2)} = 134$, actual exponent is 134 - 127 = 7.
2. base two normal form $+1.1111010111_{(2)} \cdot 2^{+7}$
1. base two number $+11111010.111_{(2)}$
0. base ten number $+250.875$

2.4.4 Special cases

As we said before, the values of 0 and 255 of the offset exponent have special meaning, they are reserved for special cases. Below is a table of what these special cases mean.

Special cases of (single precision) floating point numbers

		significant	
		all 0s	not all 0s
(offset) exponent	0 (all 0 bits)	signed zero, ± 0	subnormal numbers – see below
	255 (all 1 bits)	signed infinity, $\pm$Inf	NaN (not a number) – see below

- **subnormal numbers**: the integer part in the normal form is assumed to be 0, and the actual exponent is assumed to be -126. This allows for numbers smaller than 2^{-126} (hence the name), at the cost of some precision lost.
- **NaN (not a number)**: this is used to signal the result of undefined operations, such as Inf-Inf, or 0/0.

2.4.5 Double precision

Double precision works analogously, both for normalized numbers, as well as the special cases. For simplicity, we show the differences through a comparison table of single precision and double precision.

Floating point numbers

	single precision	double precision
stored on	4 bytes (32 bits)	8 bytes (64 bits)
sign stored on	1 bit	1 bit
exponent stored on	8 bits	11 bits
exponent offset	$127 = 2^{8-1} - 1$	$1023 = 2^{11-1} - 1$
range of actual exponent	-126 – +127	-1022 – +1023
special cases of offset exponent	$0, 255 = 2^8 - 1$	$0, 2047 = 2^{11} - 1$
subnormal exponent	-126	-1022
significant stored on	23 bits	52 bits

Chapter 3

Exercises

Back to top

Contents of chapter

3.1 Mathematics introduction

Exercise 3.1 (Floor, ceil, round and fractional part)**.** Complete the table:

x	-4	-3.8	-3.5	-3.2	-3	3	3.2	3.5	3.8	4
$\lfloor x \rfloor$										
$\lceil x \rceil$										
Round(x)										
$\{x\}$										

Exercise 3.2 (Floor, ceil, round and fractional part functions)**.** Plot the floor, ceil, round and fractional part functions, say, on the interval $[-2, 3]$!

Exercise 3.3 (div and mod operations)**.** Calculate ± 12 div ± 5 and ± 12 mod ± 5 for all possible combination of signs!

3.2 Number representations

3.2.1 Conversion between number systems

Exercise 3.4 (Number in different bases)**.** Each row should contain the same number in different bases.

a) Complete the table!

base						
2	3	4	8	10	12	16
10110110						
	120102					
		30201				
			7421			
				2024		
					8A4B	
						5F03

b) Complete the table!

base						
2	3	4	8	10	12	16
101110.011						
	12012.12					
		302.01				
			742.15			
				2024.25		
					8A4B.36	
						5F03.84

3.2.2 Number of digits

Exercise 3.5 (Number of digits)**.** Using the theorem on the number of digits, without making the conversion, predict how many digits a number requires in a different base!

a) Complete the table!

number of digits

		base		
		2	10	16
	2^2			
number	10^{10}			
	16^{16}			

b) A number is 34 digits in base two. How many digits is it in base sixteen? Can we generalize a rule?

c) A number is 9 digits in base sixteen. How many digits is it in base two? Can we generalize a rule?

3.2.3 Floating point numbers

Exercise 3.6 (Floating point numbers)**.** a) Represent the following (base ten) numbers as single precision floating point numbers! What are the resulting hexadecimal bytes?

1387.1875 – 1208.6875 625.90625

b) The following hexadecimal bytes represent single precision floating point numbers. What are the numbers (in base ten)?

```
C4 69 26 00      44 5B 0E 00      C4 BE 9A 00
```

Part II

Algorithms: representation and quantification. Recursion.

Chapter 4

Lecture

Back to top

Contents of chapter

4.1 Representations of an algorithm

We introduce some representations of an algorithm that do not rely on specific programming languages. Of course, these representations have their own conventions, that is, we can think of them as alternative languages that are not implemented to be translated for a computer, instead are meant to be more easily human-readable. These will be pseudocode and flowcharts.

4.1.1 Pseudocode and flowchart

Pseudocode looks quite like a programming language, however is not implemented. Primarily each instruction is in its own line, and it is read from top to bottom, unless the instructions say otherwise. We signal *blocks* of code with indentation; we think of blocks as units of code that belong together and are always executed together, skipped together or repeated together, such as a conditional block, or the body of a loop (that is repeated altogether).

Flowcharts are a visual representation for code, where the control flow of the program is represented by arrows. We typically draw them from top to bottom and left to right. Different instructions are written in different shapes.

We will show a few examples of both representations, but to save time and space, most of the times during this course we will write our algorithms using pseudocode.

In Table 4.1 below, we show some (corresponding) common control elements of each representation. These are the basic instructions that we can give the computer, and so we must plan our algorithms using these control elements. These are common and implemented in nearly all programming languages; some languages may have variations or additional control elements as well.

For the contents of Table 4.1 to make sense, we define some notation and functions we may use in both pseudocode and flowcharts:

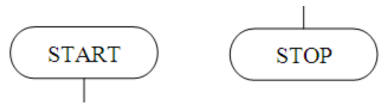
- `@` refers to the memory address of a variable.[1] When calling a procedure, we assume only copies of the variables are passed on, hence we must pass on the memory address of a variable to be able to use it as output. Another way to think of this: by default, we have read-only access to variables, any changes made to them are local to our calculations. To have read-and-write access, we must use `@var`.
- we denote value assignment by $\leftarrow$, and $=$ stands for the logical test "is equal"
- LEN[sequence] returns the length attribute of the sequence, i.e., the number of elements in it.
- INC(var) increments the variable by 1; this is a shorthand for var $\leftarrow$ var+1.[2]
- DEC(var) decreases the variable by 1; this is a shorthand for var $\leftarrow$ var-1.[3]
- we access named fields of a complex object with object[field_name].
- we access an indexed element of a sequence with an index or [], e.g. A_i or A[i].
- convention: we index sequences starting from 1.

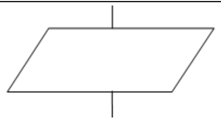
Please scroll down for Table 4.1 – it's set in landscape and starts on the next page. Note that it also spans several pages! We return to examples of algorithms below the table.

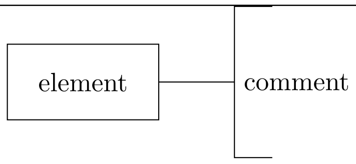
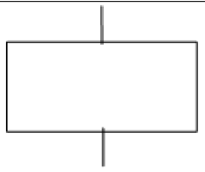
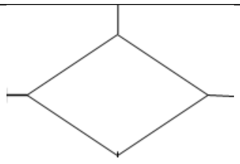
[1] In C and related languages, this corresponds to a pointer.
[2] In C, this would be `var++`.
[3] In C, this would be `var--`.

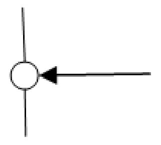
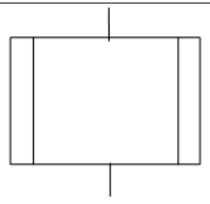
Table 4.1: Common control elements represented in pseudocode and flowchart.

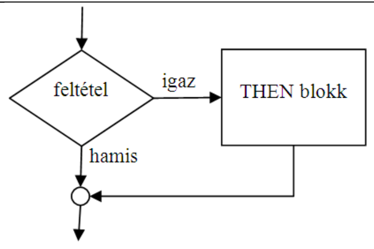
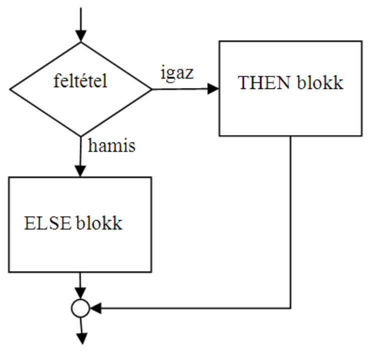
Element	**Pseudocode**	**Flowchart**

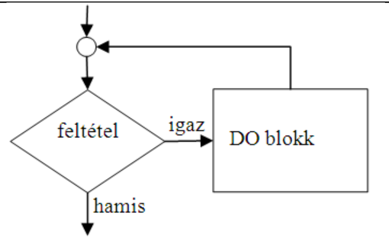
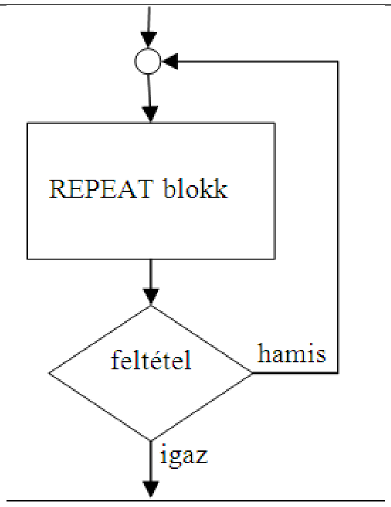
start and stop	*for program:*
	START
	...
	STOP
	for procedure:
	PROCEDURE_NAME(*input variable(s)*)
	...
	RETURN(*output variable(s)*)
		

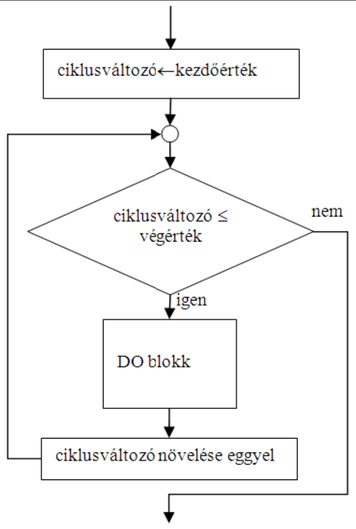
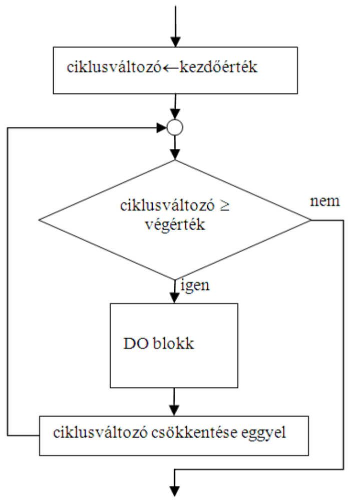
input, output	READ(@*variable*)
	
	PRINT(*variable*)
		
		with *in* or *out* in the corner

comment	// comment	
“function” / instruction	instruction, e.g. value assignment: $a \leftarrow 2$	
decision – we write TRUE and FALSE on the out arrows to indicate which to follow. building block of loops, conditions, etc.		

Element	**Pseudocode**	**Flowchart**
collection point – branches must be collected by these, every other element is only allowed one in-arrow!		
procedure call	*output variable(s)* ← PROCEDURE_NAME(*input variable(s)*)	

condition – with or without false branch	IF *condition*
	    THEN *block if true*
	(ELSE *block if false*)
		 

Element	**Pseudocode**	**Flowchart**
while loop – while condition is true, repeat loop body	WHILE *condition* DO *cycle body, block to repeat*	
repeat until loop – repeat loop body until condition becomes true	REPEAT *cycle body, block to repeat* UNTIL *condition*	

Element	Pseudocode	Flowchart
for loop – repeat loop body for increasing or decreasing sequence of values	FOR $i \leftarrow 1$ TO n DO *loop body, block to repeat* FOR $i \leftarrow n$ DOWNTO 1 DO *loop body, block to repeat*	 ciklusváltozó←kezdőérték ciklusváltozó ≤ végérték nem igen DO blokk ciklusváltozó növelése eggyel  ciklusváltozó←kezdőérték ciklusváltozó ≥ végérték nem igen DO blokk ciklusváltozó csökkentése eggyel

4.1.2 Examples

We show some examples to demonstrate two things at once. We show how the same problem can be solved by different algorithms. We also show the representation of the same algorithm in pseudocode as well as flowchart form.

The problem to solve: We consider the problem of finding the maximal element of a sequence (of numbers). We do not yet care about the exact implementation of the data, but we assume we can access each element directly by its index.[4]

We consider the following algorithms:

- an iterative algorithm: we read the sequence from left to right, always storing the current record. We can also consider this an example of the *dynamic programming* principle – the subproblems are *nested* like matryoshka dolls, we need the answer for one to be able to solve the next.
- a recursive algorithm: we split the sequence in half, find the maximum of each half, then take the larger of the two. For the partial sequences, we repeat the same, until we have a sequence of one element only. We can also consider this an example of the *divide and conquer* principle – we split the problem into *independent* subproblems, then we combine the partial solutions. Note that we guarantee the recursion is finite by splitting the sequence in half, thus reducing the problem size, and handling the trivial case of a 1-element sequence.

Algorithm 4.1 Iterative maximum with pseudocode

```
1: MAX_ITER(A,@max)
2: // INPUT: sequence A
3: // OUTPUT: max, the maximal element of A
4: max ← -Inf
5: FOR i ← 1 TO LEN[A] DO
6:     IF A[i] > max THEN
7:         max ← A[i]
8: RETURN(max)
```

Algorithm 4.2 Recursive maximum with pseudocode

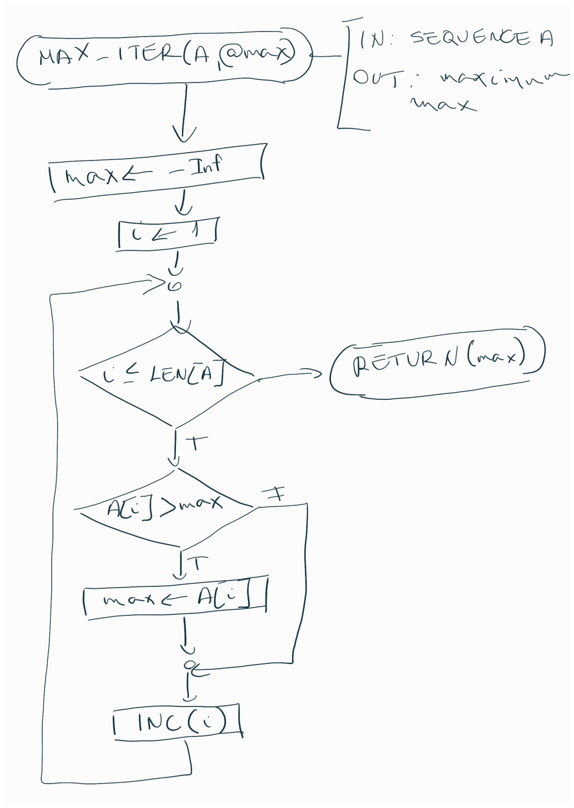
```
1: MAX_ITER(A,a,b,@max)
2: // INPUT: sequence A, indices $1 \leq a \leq b \leq$ LEN[A]
3: // OUTPUT: max, the maximal element of (A[a], ..., A[b])
4: IF a = b THEN      // maximum of one element sequence is itself
5:     max ← A[a]
6: ELSE
7:     c ← $\lfloor \frac{a+b}{2} \rfloor$      // calculate midpoint
8:     $m_1$ ← MAX_REC(A,a,c,@$m_1$)
9:     $m_2$ ← MAX_REC(A,c+1,b,@$m_2$)
10:    max ← max{$m_1, m_2$}
11: RETURN(max)
```

We demonstrate these same algorithms as flowcharts in Figure 4.1.

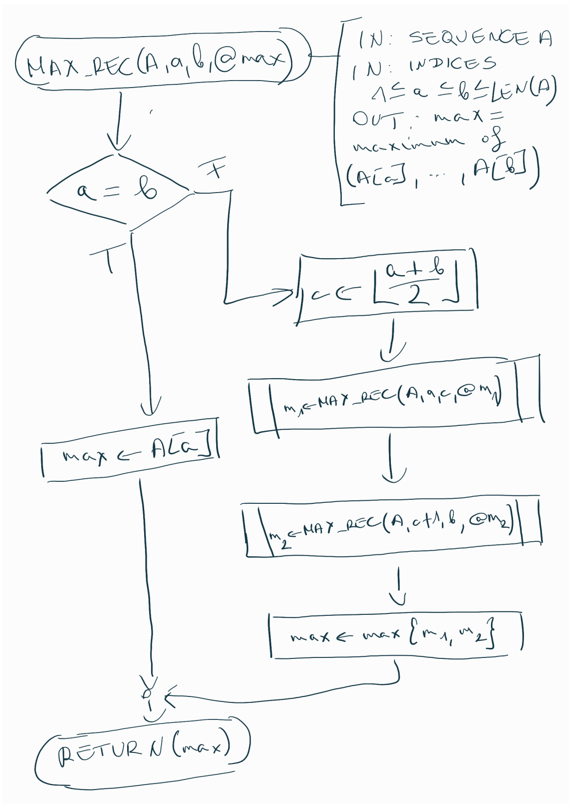
[4] In C, this would mean an array. In Python, it might also mean a list.

Figure 4.1: Flowchart of Algorithms 4.1, 4.2

(a) Flowchart of MAX_ITER in Algorithm 4.1



(b) Flowchart of MAX_REC in Algorithm 4.2



4.2 Qualitative description of an algorithm

The following concepts are used to measure the efficiency of an algorithm, in terms of the runtime, as well as the memory usage. The "complexity" definitions focus on the worst case scenario, and still try to make it as low/efficient as possible. In most cases, this is the most relevant metric; and in many cases, there isn't much of a difference between typical and worst case scenarios. But rarely, we might instead consider the efficiency of the algorithm in the typical/average case (an example in this course will be the Quicksort algorithm, coming later).

In most cases, the runtime will be our most important metric; however, when working with large data, we might have to consider memory usage as well, because if we run out of memory, the running of the program will be aborted.

4.2.1 The metrics

Definition 4.1 (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 4.2 (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 4.3 (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

$$T_A(n) := \max_{\substack{x \in D \\ |x| \leq n}} t_A(x), \tag{4.1}$$

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

Definition 4.4 (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

$$S_A(n) := \max_{\substack{x \in D \\ |x| \leq n}} s_A(x), \tag{4.2}$$

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

4.2.2 Why do we care about time complexity of an algorithm?

Why do we care about time complexity $T_A(n)$ of an algorithm? It greatly impacts the runtime of the algorithm/program, and how large of an input we can handle within reasonable time frames.

We make a table below for demonstration.

We shall denote the input size by n, and consider various time complexity functions $f(n) = T_A(n)$, that is, the (maximum) number of operations m required to run algorithm A on an input of size (at most) n. If we know the number of operations required is $m = f(n)$, we can rearrange the equation and "reverse engineer" $n = f^{-1}(m)$, the input size that we can process with the given number of operations m. For example, if $m = \log_2(n)$, then $n = 2^m$.

We prepare Table 4.2 where:

- suppose we have a computer whose processor can make 10^6 operations a second,
- the columns will be labeled by different time intervals (which indirectly determine the number m of operations available),
- the rows will be labeled by various $m = f(n) = T_A(n)$ time complexity functions (we'll also write $m = f^{-1}(n)$, where it can be expressed explicitly)
- the cells will show the largest input size n (measured in bits) that we can process within the given time with the given time complexity.

Again, Table 4.2 is typeset in landscape, please scroll down to the next page.

Table 4.2: Maximum input size processed in given time with various time complexity functions

time	1 second	1 minute	1 hour	1 day	1 month (30 days)	1 year (12 months)	1 century (100 years)
operations m	10^6	$6 \cdot 10^7$	$3.6 \cdot 10^9$	$8.64 \cdot 10^{10}$	$2.59 \cdot 10^{12}$	$3.108 \cdot 10^{13}$	$3.108 \cdot 10^{15}$
$m = f(n) = \log_2(n)$ $n = 2^m$	$2^{10^6} \approx 10^{18}$	$6.4 \cdot 10^{22}$			$\dots$		
$m = f(n) = \sqrt{n}$ $n = m^2$	10^{12}	$3.6 \cdot 10^{15}$	$1.296 \cdot 10^{19}$	$7.465 \cdot 10^{21}$		$\dots$	
$m = f(n) = n$	10^6	$6 \cdot 10^7$	$3.6 \cdot 10^9$	$8.64 \cdot 10^{10}$	$2.59 \cdot 10^{12}$	$3.108 \cdot 10^{13}$	$3.108 \cdot 10^{15}$
$m = f(n) = n^2$ $n = \sqrt{m}$	10^3	$7.74 \cdot 10^3$	$6 \cdot 10^4$	$2.93 \cdot 10^5$	$1.61 \cdot 10^6$	$5.57 \cdot 10^6$	$5.57 \cdot 10^7$
$m = f(n) = n^3$ $n = \sqrt[3]{m}$	10^2	$3.9 \cdot 10^2$	$1.53 \cdot 10^3$	$4.42 \cdot 10^3$	$1.37 \cdot 10^4$	$3.14 \cdot 10^4$	$1.46 \cdot 10^5$
$m = f(n) = 2^n$ $n = \log_2(m)$	≈ 20	≈ 26	≈ 32	≈ 36	≈ 41	≈ 45	≈ 51
$m = f(n) = n!$ no closed form for $n = f^{-1}(m)$	≈ 9	≈ 11	≈ 12	≈ 13	≈ 15	≈ 16	≈ 17

Reminder:
- 1 byte = 8 bits $\approx$ 10 bits
- 1 kB (kilobyte) = 1024 bytes $\approx 10^4$ bits
- 1 MB (megabyte) = 1024 kB $\approx 10^7$ bits
- 1 GB (gigabyte) = 1024 MB $\approx 10^{10}$ bits
- 1 TB (terabyte) = 1024 GB $\approx 10^{13}$ bits
- more...

4.3 Growth rates, the “master” theorem and Fibonacci numbers

4.3.1 Describing growth rates

Calculating the number of operations required for an algorithm, we may end up with a rather complicated time complexity function, such as $T_A(n) = \frac{3}{2}n^2 + 2n\log(n) + \frac{1}{2}n$. However, for the runtimes, we are only interested in the leading order term (the largest one). The growth order more or less focuses on the fastest growing term, also ignoring constant multipliers. The below definitions provide a precise mathematical way to describe the growth of such functions with various terms that grow at different rates.

Definition 4.5 (Growth rates, *order* of functions)**.** 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 4.6 (Some common growth rates)**.**

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 $a > 1$

This will help us cut away the “fluff”, aka the non-leading order terms, but also the constants. We do not particularly care about constant factors. While a program taking twice as long to run does matter, but it matters much less than how the runtime changes if we double the input size. Let us show some examples:

- for $T(n) = n$, aka linear complexity, double input size means double runtime
- for $T(n) = n^2$, aka quadratic complexity, double input size means *four times* the runtime

- for $T(n) = 2^n$, aka exponential complexity, the increase in runtime is *explosive*, we cannot simply describe it with a multiplicative factor. Instead we give an example. Start from what we know: the number of operations needed is squared, because $2^{2n} = (2^n)^2$. Suppose the program ran in 1 second for input size n, supposing 10^6 operations per second on the processor, that translates to 10^6 operations. For input size $2n$, we then have 10^{12} operations, which will take 10^6 seconds, which is nearly 12 days (!).

Statement 4.1 (Order of magnitude of logarithm)**.**

1. $\log(n)$ is a growing function: $\log(n) = \omega(1)$
2. however it grows slower than any polynomial, for any $\varepsilon > 0$, $\log(n) = o(n^\varepsilon)$

Proof.

1. clearly, $\frac{\log(n)}{1} \to \infty$.
2. as before, $\lim_{n\to\infty} \log(n) = \infty$ and for $\varepsilon > 0$, also $\lim_{n\to\infty} n^\varepsilon = \infty$. To calculate $\lim_{n\to\infty} \frac{\log(n)}{n^\varepsilon}$, we use L'Hospital's rule and calculate

$$\lim_{n\to\infty} \frac{(\log(n))'}{(n^\varepsilon)'} = \lim_{n\to\infty} \frac{\frac{1}{n}}{\varepsilon \cdot n^{\varepsilon-1}} = \lim_{n\to\infty} \frac{1}{\varepsilon \cdot n^\varepsilon} = 0.$$

Hence $\lim_{n\to\infty} \frac{\log(n)}{n^\varepsilon}$ also exists and is equal to 0, by definition, $\log(n) = o(n^\varepsilon)$.

□

The definitions below are needed a little bit later, for the upcoming "master" theorem.

Definition 4.7 (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 4.8 (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 $f(n) = O\big(n^{p-\varepsilon}\big)$.

Note that, by Statement 4.1, $\log(n)$ grows faster than constant, but does *not* grow polynomially faster than constant.

4.3.2 Growth rate of a recursive algorithm

For a recursive algorithm, sometimes we cannot directly describe its time complexity, but we can express it in terms of the same time complexity function for smaller inputs. In mathematical speak, we can write a recursive equation for it:

Definition 4.9 (Recurrence equation)**.** A recurrence 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 $k_i < n$.

4.3.2.1 The "master" theorem

In some cases, a recursive algorithm splits the problem into several subproblems whose size is a *fraction* of the original. In such cases, the "master" theorem below may help us determine the time complexity of the algorithm. This is a theorem that may help determine the growth order of a function which is not known explicitly, but satisfies a recurrence equation. However, it isn't omnipotent. It doesn't give an explicit solution, and there are some cases where it cannot even determine the growth order. In cases when we can use the "master" theorem to calculate the time complexity of a recursive algorithms, we often find logarithmic or polynomial growth.

Theorem 4.1 (The "master" theorem)**.** Suppose we have a recurrence 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 $\left\lfloor \frac{n}{b} \right\rfloor$ or $\left\lceil \frac{n}{b} \right\rceil$ 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
$$T(n) = \Theta\big(g(n)\big). \tag{4.3}$$
2. If $f(n) = \Theta\big(g(n)\big)$, then
$$T(n) = \Theta\big(g(n) \cdot \log(n)\big). \tag{4.4}$$
3. Suppose $f(n)$ grows *polynomially faster* than $g(n)$, as well as the so-called *regularity condition* holds for f, that is,
$$\text{there exists } c < 1, c \in \mathbb{R}^+ \text{ and } n_0 \in \mathbb{N} \text{ such that for all } n \geq n_0, a \cdot f\left(\frac{n}{b}\right) \leq c \cdot f(n). \tag{4.5}$$
If both above conditions hold, then
$$T(n) = \Theta\big(f(n)\big). \tag{4.6}$$

The proof is beyond the scope of this course. However, let us show why the test polinomial $g(n = n^p)$, with the specific $p = \log_b(a)$ is important.

Suppose we have a recursion of the form

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

with $a \geq 1$ and $b > 1$ as in the "master" theorem, however without the $f(n)$ term. We show, simply by substitution, that $T(n) = g(n) = n^p$ is a solution of the recursion.

$$\begin{aligned}
n^p = g(n) = T(n) &\overset{?}{=} a \cdot T\left(\frac{n}{b}\right) = a \cdot g\left(\frac{n}{b}\right) = a \cdot \left(\frac{n}{b}\right)^p \\
n^p &\overset{?}{=} a \cdot \frac{n^p}{b^p} \\
b^p &\overset{?}{=} a \\
b^{\log_b(a)} &\overset{?}{=} a \\
a &\overset{?}{=} a
\end{aligned}$$

The last line is clearly an identity, hence the first line holds as well.

4.3.3 Growth rates of the example algorithms

Take the iterative Algorithm 4.1.

- for simplicity, let us denote n = LEN[A], and let us calculate the number of operations in terms of this n.[5]
- for an iterative algorithm, the number of operation required is straightforward to calculate. we have:
  - a single assignment: 1 operation
  - a FOR loop with n repetitions
    - a comparison and maybe an assignment, at most 2 operations
    - management of the FOR loop itself, as demonstrated in Figure 4.1a, takes an additional condition check and incrementation of the index variable, 2n operations
  - a return command, 1 operation
- in total at most 4n+2 operations, in terms of n = LEN[A]. Hence Algorithm 4.1 has time complexity $T_A(n) = O(n)$.
- the number of bits required to store A is a constant multiple of n, depending on which number data type is used, for example with unsigned integers on 1 byte, this is 8n, for single precision floating point numbers, this is 32n. In terms of the bits used, we may get b/2 + 2, or b/8 + 2 operations as an upper bound, and we still get an $T_A(b) = O(b)$ time complexity. Hence why we often neglect such constant factors.

Now take the recursive Algorithm 4.2.

- once again, let us calculate in terms of the length of the sequence, which we can now describe as $n = b - a + 1$.
- for a recursive algorithm, we will get a recursive equation for the time complexity.
- we have:
  - a condition check, 1 operation
  - on the TRUE branch, a single assignment = 1 operation
  - on the false branch:
    - an assignment, 1 operation
    - two recursive calls for sequences of size at most $\lceil \frac{n}{2} \rceil$
    - finding the maximum of two elements and assigning it, 2 operations
  - on both branches, a return command.
- we get a recursive upper bound:
$$T(n) \leq 2T\left(\left\lceil \frac{n}{2} \right\rceil\right) + 5$$
- we try to identify the growth order using the "master" theorem. We have test polynomial $g(n) = n^1$ and $f(n) = 5$, which has polynomially slower growth, this is case 1. of the theorem. Hence we get $T(n) = \Theta(g(n)) = \Theta(n)$.

[5]The actual input size in bits is some constant times n – for example, if we have integers on 1 byte, it's $8n$ bits. The ordo notation doesn't care about multiplicative constants, so we can do this simplification and still get the right growth order. In the following, we will often make similar simplifications.

4.3.4 Fibonacci numbers, Fibonacci recursion

Definition 4.10 (Fibonacci numbers)**.** The Fibonacci numbers is a number series defined recursively as

$$\left.\begin{array}{r} F_0 := 0 \\ F_1 := 1 \end{array}\right\} \quad \text{initial conditions}$$
$$n \geq 2: \; F_n := F_{n-1} + F_{n-2} \qquad \text{recursive condition} \tag{4.7}$$

The first few numbers are

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, \ldots$$

Above, we have seen the "master theorem", which can be applied when a recursive algorithm reduces the problem to subproblems whose size is a *fraction* of the original problem size. However, sometimes we can only reduce the problem size *by a constant*.[6] That is when the time complexity $T(n)$ satisfies a recursive equation similar to the definition of the Fibonacci numbers: the previous terms in the formula aren't a fraction of the original, but only smaller by a subtracted constant. In such cases, $T(n)$ often grows exponentially.

4.3.4.1 Growth of Fibonacci numbers

The theorem below gives a closed form, direct calculation for F_n for any n.

Theorem 4.2 (Binet's formula)**.** For any $n \in \mathbb{N}$, element F_n of the Fibonacci series can be obtained directly by the formula

$$F_n = \frac{1}{\sqrt{5}} \left(\Phi^n - \overline{\Phi}^n \right), \tag{4.8}$$

where

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

Proof. Let us look for a solution in the form $F_n = x^n$. The recursion condition then becomes

$$x^n = x^{n-1} + x^{n-2},$$

or equivalently,

$$x^2 = x + 1. \tag{4.9}$$

We rearrange this into a quadratic equation

$$x^2 - x - 1 = 0,$$

which has solutions

$$x_{1,2} = \frac{+1 \pm \sqrt{5}}{2}.$$

Denote

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

We refine the form of the solution as $a\Phi^n + b\bar{\Phi}^n$. As both Φ and $\overline{\Phi}$ satisfy (4.9), the recursion condition is still satisfied. We use the initial conditions to calculate the constants a and b:

$$a \cdot \Phi^0 + b \cdot \overline{\Phi}^0 = F_0 = 0$$
$$a \cdot \Phi^1 + b \cdot \overline{\Phi}^1 = F_1 = 1,$$

[6]This may happen for example in some graph problems where cases are split based on whether the nth vertex is involved.

then

$$\begin{aligned} a+b&=0\\ a\cdot\frac{1+\sqrt{5}}{2}+b\cdot\frac{1-\sqrt{5}}{2}&=1. \end{aligned}$$

From the first equation, we have $b=-a$, which we substitute into the second:

$$\begin{aligned} a\cdot\frac{1+\sqrt{5}}{2}-a\cdot\frac{1-\sqrt{5}}{2}&=\\ =\frac{2a\sqrt{5}}{2}&=1, \end{aligned}$$

that is, $a=\frac{1}{\sqrt{5}}$, and thus $b=-\frac{1}{\sqrt{5}}$. □

This proof of Binet's formula shows how we can calculate the exact growth rate function, even if we may have different constants, for example a recursion like

$$T_A(n)=T_A(n-1)+2\cdot T_A(n-2)$$

can be solved, it only leads to a different quadratic equation, but the constants can again be calculated from the initial conditions.

If the recursion relies on $T_A(n-3)$, we get a cubic equation for x_i, and three roots; if even more previous terms are there, we will have a higher order polynomial equation. Finding the roots becomes harder in general (if analytic methods fail, still possible with numeric methods), but in theory, a solution in the form $\sum_i a_i x_i$ is always possible.

Based on Binet's formula, there is a simpler way to calculate the Fibonacci numbers.

Theorem 4.3 (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

$$F_n=\text{Round}\left(\frac{1}{\sqrt{5}}\Phi^n\right). \tag{4.10}$$

Proof. This follows from Binet's formula, as $-\frac{1}{\sqrt{5}}\overline{\Phi}^n$ is exponentially decreasing and indeed becomes a rounding error. □

4.4 Overview

In this chapter, we learnt about time **complexity of** an algorithm. This means, for any input on n bits, it takes at most $T(n)$ processor operations to get our result (worst case scenario).

Why is this important? Because the number of processor operations is proportional to the **runtime**, as the processor can do a fixed number of processor operations in each second. (The number of operations per second can differ by computer, so calculating the total number of operations is a more universal metric.) When we have different algorithms for the same problem, we'd like to pick the faster one – we don't have centuries to wait for results from a very silly program, after all. So we compare them using time complexity.

The **ordo notation** is meant to simplify the exact time complexity, focusing on the speed of growth. This is meant to measure how the algorithm scales for large inputs. We most often use the big ordo (”tight upper bound”), or the theta (exact growth rate, aside from constant multiplier). When some algorithms have the same growth rate, e.g. both are $O(n)$, we begin to care about constant factors too.

We also talked about **ways to calculate** $T(n)$, based on the type of algorithm we have.

- for an **iterative algorithm**, we can directly calculate $T(n)$ – though most often we only focus on the growth rate.
- for **recursive algorithms**, we get a recurrence equation for $T(n)$ – it is given in terms of previous values of the function (other $T(k)$ terms, for $k < n$, possibly for several different k). There are two *substantially* different types:
  - **Master theorem type:** we reduce the problem to subproblems whose size is a *fraction* of the original, e.g. $T(n) = 2T\left(\frac{n}{2}\right) + f(n)$. This often leads to polynomial growth, that is, $T(n) = \Theta(n^p)$ for some $p \in \mathbb{R}^+$ (just like the test polinomial in the Master theorem 4.1.)
  - There are exceptions. For example, we can apply (the second case of) the Master theorem to $T(n) = T\left(\frac{n}{2}\right) + C$ and get $T(n) = O(\log_2 n)$ (binary search algorithm, coming soon).
  - **Fibonacci type:** we reduce the problem to subproblems whose size is only *smaller by a constant*, e.g. $T(n) = T(n-1) + T(n-2)$. Like in Binet's formula (Theorem 4.2), this usually leads to exponential growth, that is, $T(n) = \Theta(\alpha^n)$, for some $\alpha \in \mathbb{R}^+$.
  - There are exceptions again. For example, $T(n) = T(n-1) + C$ (arising from Exercise 6.6) looks like the Fibonacci type, but we can prove by induction that the solution is $T(n) = O(n)$.

Chapter 5

Instruction

Back to top

Contents of chapter

5.1 Control flow, stack

Like us humans can outsource and delegate tasks, computer programs may also run other programs (or a "clone" of itself) to do the same.

When a procedure is activated by the main program or another procedure, we call that a *procedure call.* A program/procedure can also call itself, that creates a "clone" or copy in the computer. We call it another running *instance* of the program. When a procedure is called by itself, we call it a *recursive call.*

5.1.1 Recursion and stack

For now, ignore the fact that modern computers have several processors and can handle multiple threads and calculations at once, and imagine the computer can only run one thing at a time. When a procedure is called, the previously running program or procedure must be halted, and control is given over to the newly called procedure. The analogue would be, when a manager delegates part of a task, they need to wait for their subordinate to finish before the manager themselves can proceed.

So that control flow can be tracked, and control can be given back later, there is a special part of the memory called the *stack.* When a procedure is called, an entry is made in the stack, with the input and output variables, as well as the memory address of the line of code where the previously running program or procedure can be resumed. Think of the stack as a company-wide shared TODO-list within the computer's memory.

It is possible the procedure then calls another procedure, just like an employee might recieve a task from their manager, then delegate a part of it to an intern. Now the manager is waiting on the empoyee, who is waiting on the intern to finish. Each delegated task becomes a new entry in the stack below the previous one, we say the stack becomes deeper. Once the called procedure finishes running, the output variables are returned to the program unit that called it, and control is transferred back as well, and the entry of the stack is deleted.

The analogue breaks down here a bit, since several employees or interns can work in parallel on their assigned subtasks. On the computer, assuming a single computing unit, only one task can run at once. This is like the employee handing out one subtask at a time, then once gets it back from their intern, only then do they assign another intern another subtask. Only when the employee's task is finished and returned to the manager, does the manager assign another employee another task, and so on. On the TODO-list, we only see the task from the manager, and one subtask to one intern at a time. That intern's TODO is then deleted before the other intern's TODO is added. When we are the same level of the delegation hierarchy, they appear in the same spot of the great shared TODO-list at different times. On the computer, we call this the level or depth of the stack.

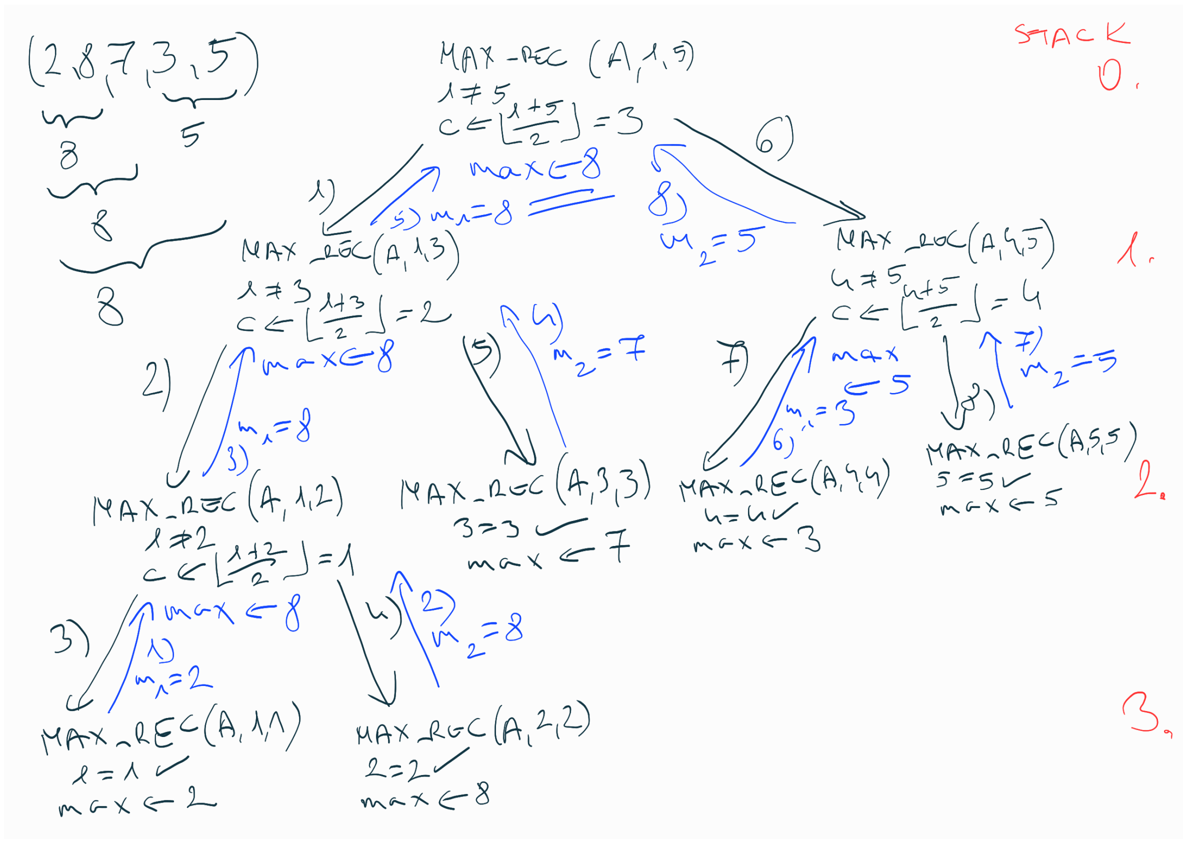
5.1.2 Recursive call tree

We can visually represent the structure of calls made by a program or procedure for a given input by a so-called *call tree*. The analogy would be the delegation hierarchy. At the top center, we write the main program or procedure, with its input. Procedures called by it are written one level below, from left to right in the order of being called. To differentiate, we write the inputs for each procedure call. The call itself is represented by a down arrow from one procedure to the other. Procedures on the same level of the tree also occupy the level in the stack at different times. When procedures call new procedures, the stack deepens, and so does the current branch of the tree. We often represent return of control by upwards arrows. We can additionally write the order of calls on the down arrows, and the up arrows may also have the return value written on them. We are often interested in the maximum depth of the stack used, i.e., the height of this tree. Note that the main program/procedure call is at depth 0.

We can thus deduce the evolution of the stack and the completion order of subtasks from this tree. We go down and left as far as we can, then when we reach the bottom, we backtrack once – first subtask collected. Again try to go on the leftmost downwards path that hasn't been travelled yet to collect the next subtask. If there's no such thing, the task ca be handed back to the manager, backtrack and go up one level again. Continue until all paths are travelled and all subtasks are collected and merged into the main task.

We give an example of a recursive call tree. Consider the sequence A = (2,8,7,3,5), and suppose we want to find its maximum using the recursive Algorithm 4.2 (from the lecture part). Clearly, the initial call must be MAX_REC(A,1,5,@max). We draw the recursive call tree in Figure 5.1.

Figure 5.1: Recursive call tree for Algorithm 4.2 on a 5 element sequence



Chapter 6

Exercises

Back to top

Contents of chapter

6.1 Ordo notation and the “master” theorem

6.1.1 Ordo notation

Exercise 6.1 (Relations between various ordo notation)**.** Prove the following statements:

a) If $f(n) = o\big(g(n)\big)$, then also $f(n) = O\big(g(n)\big)$.

b) Is part a) reversible? Why, or why not?

c) If $f(n) = \omega\big(g(n)\big)$, then also $f(n) = \Omega\big(g(n)\big)$.

d) Is part c) reversible? Why or why not?

e) If $f(n) = O\big(g(n)\big)$ and $f(n) = \Omega\big(g(n)\big)$, then $f(n) = \Theta\big(g(n)\big)$.

f) Is part e) reversible? Why or why not?

Exercise 6.2 (Polynomials)**.** Are the following statements true?

a) $2n^2 + 3n = O(n^2)$

b) $2n^2 + 3n = o(n^2)$

c) $2n^2 + 3n = O(n^3)$

d) $2n^2 + 3n = O(n)$

e) $2n^2 + 3n = \Omega(n^2)$

f) $2n^2 + 3n = \omega(n^2)$

g) $2n^2 + 3n = \Omega(n^3)$

h) $2n^2 + 3n = \Omega(n)$

i) $2n^2 + 3n = \Theta(n^2)$

Exercise 6.3 (Logarithm)**.** Are the following statements true?

a) $\log(n) = O(n)$

b) $\log(n) = o(n)$

c) $\log(n) = O(n^\varepsilon)$ for any or all $\varepsilon \in \mathbb{R}^+$

d) $\log(n) = o(n^\varepsilon)$ for any or all $\varepsilon \in \mathbb{R}^+$

e) $\log(n) = \Omega(n^\varepsilon)$ for any or all $\varepsilon \in \mathbb{R}^+$

f) $\log(n) = \omega(n^\varepsilon)$ for any or all $\varepsilon \in \mathbb{R}^+$

g) Is $\log(n)$ polynomially faster growing than $1 = n^0$?

h) Is $\log(n)$ polynomially slower growing than n?

6.1.2 Application of the "master" theorem

Exercise 6.4 ("Master" theorem)**.** Consider the following, recursively defined functions. Can we determine their growth order using the master theorem? If yes, what is their growth order?

a) $T(n) = 2 \cdot T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + \log(n)$

b) $T(n) = 2 \cdot T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + 2n + 3$

c) $T(n) = 2 \cdot T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + n \cdot \log(n)$

d) $T(n) = 2 \cdot T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + n^{3/2}$

6.2 Recursion

6.2.1 Factorial in various ways

For $n \in \mathbb{Z}^+$, we define n factorial,

$$n! := 1 \cdot 2 \cdot \ldots \cdot n,$$

the product of positive integers up to n.

Exercise 6.5 (Factorial 1)**.** Write a procedure that calculates the factorial straightforwardly, simply by multiplying the numbers from 1 to n. We can think of it as a dynamic programming or iterative approach. What is the time complexity of this algorithm?

Exercise 6.6 (Factorial 2)**.** We could use an alternate, recursive definition for the factorial:

$$n! = \begin{cases} (n-1)! \cdot n, & \text{if } n > 1, \\ 1, & \text{if } n = 1. \end{cases}$$

Write a procedure that calculates the factorial based on this recursive formula. What is the time complexity of this algorithm? Suppose the procedure is initially called to calculate 5!. Draw the recursive call tree! What is the largest depth? How many *recursive* calls are made?

Exercise 6.7 (Factorial 3)**.** More generally, let us define the product of consecutive numbers from $m \leq n \in \mathbb{Z}^+$ as

$$P(m,n) := \begin{cases} m, & \text{if } m = n, \\ P\left(m, \left\lfloor \frac{m+n}{2} \right\rfloor\right) \cdot P\left(\left\lfloor \frac{m+n}{2} \right\rfloor + 1, n\right), & \text{if } m < n. \end{cases}$$

Write a procedure that calculates the factorial based on this recursive formula. What is the time complexity of this algorithm? Suppose the procedure is initially called to calculate $5! = P(1,5)$. Draw the recursive call tree! What is the largest depth? How many *recursive* calls are made?

6.2.2 Binomial coefficient

Exercise 6.8 (Binomial coefficient)**.** For $k \leq n \in \mathbb{N}$, the binomial coefficient "n choose k" is defined recursively as

$$\binom{n}{k} := \begin{cases} 1, & \text{if } k = 0, \\ 1, & \text{if } k = n, \\ \binom{n-1}{k-1} + \binom{n-1}{k}, & \text{if } 0 < k < n. \end{cases}$$

Write a procedure that calculates $\binom{n}{k}$ based on this recursion. What is the time complexity of this algorithm? Suppose it is called to calculate $\binom{4}{3}$. Draw the recursive call tree! What is the largest depth? How many *recursive* calls are made?

6.2.3 Fibonacci numbers

Recall the Fibonacci numbers defined in Definition 4.10.

Exercise 6.9 (Fibonacci)**.** Write a procedure that calculates the Fibonacci number $F(n)$ based on the recursive formula. What is the time complexity of this algorithm? Suppose it is called to calculate $F(5)$. Draw (part of) the recursive call tree! Optionally: what is the largest depth, and how many *recursive* calls are made?

Exercise 6.10 (Fibonacci in other ways)**.**

a) Suppose there is a smart programmer who realizes the Fibonacci numbers could be calculated simply in sequence, and writes that as a procedure. What would be the time and storage complexity of this procedure?

b) Suppose there is an even smarter programmer who knows Binet's formula, and writes a procedure to calculate the Fibonacci numbers with this formula. What is the time complexity of this procedure?

Part III

Algorithms from number theory

Chapter 7

Lecture

Back to top

Contents of chapter

7.1 Introduction, concepts

Definition 7.1 (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 7.2 (Prime number)**.** We call an integer $p > 1$ a prime if its only positive divisors are 1 and p itself.

Theorem 7.1 (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 7.3 (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 7.4 (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 7.2 (Properties of the (common) divisor)**.** Let $d \in \mathbb{Z}$ be a common divisor of $a, b \in \mathbb{Z}$. Then

1. $|d| \leq |a|$, unless $a = 0$.
2. if both $d|a$ and $a|d$, necessarily $d = \pm a$.
3. d is also a divisor of any linear combination of a and b, i.e., for any $s \in L(a, b)$, $d|s$.

7.2 The greatest common divisor

Definition 7.5 (The greatest common divisor)**.** For $a, b \in \mathbb{Z}$, we define their greatest common divisor as follows:

$$d^\star = \gcd(a,b) := \begin{cases} 0, & \text{if } a = b = 0, \\ \max\limits_{\substack{d|a \\ d|b}} d, & \text{otherwise.} \end{cases} \tag{7.1}$$

Definition 7.6 (Relative primes)**.** We say that $a, b \in \mathbb{Z}$ are relative primes if $\gcd(a,b) = 1$.

Theorem 7.3 (Elementary properties of the greatest common divisor)**.** Let $a, b \in \mathbb{Z}$ and $d^\star = \gcd(a,b)$. Then

1. $1 \le d^\star \le \min\{|a|, |b|\}$ (unless $a = b = 0$, since then $d^\star = 0$ as well).
2. $\gcd(a,b) = \gcd(b,a) = \gcd(a,-b) = \gcd(|a|,|b|)$.
3. $\gcd(a,0) = a$.
4. for $k \in \mathbb{Z}$, $\gcd(a, k \cdot a) = a$.
5. if d is also a common divisor of a and b and $d^\star \neq 0$, then $d \le d^\star$.
6. 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 7.4 (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:

$$d^\star = \gcd(a,b) = \min_{\substack{s \in L(a,b) \\ s > 0}} s =: s^\star = x^\star \cdot a + y^\star \cdot b, \tag{7.2}$$

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

Proof. We prove $d^\star = s^\star$ by showing that both $d^\star \le s^\star$ and $s^\star \le d^\star$.

By properties of the greatest common divisor, it divides all linear combinations. In particular, $d^\star | s^\star$, and $s^\star > 0$, necessarily $d^\star \le s^\star$.

Let us now show $s^\star \le d^\star$ by showing that $s^\star$ is a common divisor. We demonstrate $s^\star | a$, the proof for b is analogous. By theorem 7.1, write $a = q \cdot s^\star + r$, with $q, r \in \mathbb{Z}$ and $0 \le r < s^\star$. We can rearrange

$$\begin{aligned} r &= a - q \cdot s^\star \\ &= a - q \cdot (x^\star a + y^\star b) \\ &= (1 - q \cdot x^\star) \cdot a + (-q \cdot y^\star) \cdot b, \end{aligned}$$

that is, $r \in L(a,b)$. We also have $0 \le r < s^\star$. As $s^\star$ is the smallest positive element of $L(a,b)$, necessarily $r = 0$, which means that $a = q \cdot s^\star$, with no remainder, i.e., $s^\star | a$. Analogously, $s^\star | b$, and by properties of the greatest common divisor, $s^\star \le d^\star$.

We can thus conclude $d^\star = s^\star$. □

Corollary 7.1. For any common divisor d, $d | d^\star$.

Proof. Since $d | s$ for any $s \in L(a,b)$, in particular $d | s^\star = d^\star$. □

Corollary 7.2. For any $n \in \mathbb{N}$,

$$\gcd(n \cdot a, n \cdot b) = n \cdot \gcd(a,b).$$

Proof. For $n = 0$, the statement is trivial. Let us consider $n > 0$. Using the representation theorem,

$$\gcd(n \cdot a, n \cdot b) = \min_{\substack{s \in L(n \cdot a, n \cdot b) \\ s > 0}} s = s_1^\star = x^\star \cdot (n \cdot a) + y^\star \cdot (n \cdot b) = n \cdot (x^\star \cdot a + y^\star \cdot b). \tag{7.3}$$

To show $x^\star \cdot a + y^\star \cdot b = \gcd(a, b)$, it is enough to show

$$x^\star \cdot a + y^\star \cdot b = s_2^\star = \min_{\substack{s \in L(a,b) \\ s > 0}} s.$$

We argue by contradiction. Clearly, $x^\star \cdot a + y^\star \cdot b \in L(a, b)$. Since

$$s_1^\star = x^\star \cdot (n \cdot a) + y^\star \cdot (n \cdot b) > 0,$$

dividing by $n > 0$, $x^\star \cdot a + y^\star \cdot b > 0$ as well. However, suppose that $s_2^\star$ is given by another linear combination

$$0 < s_2^\star = \hat{x} \cdot a + \hat{y} \cdot b < x^\star \cdot a + y^\star \cdot b.$$

Then, by $n > 0$, we also have

$$0 < \hat{x} \cdot (n \cdot a) + \hat{y} \cdot (n \cdot b) < x^\star \cdot (n \cdot a) + y^\star \cdot (n \cdot b) = s_1^\star.$$

That is, we have found a positive element of $L(n \cdot a, n \cdot b)$ smaller than $s_1^\star$, which is not possible. Hence indeed $\gcd(a, b) = s_2^\star = x^\star \cdot a = y^\star \cdot b$. Then by (7.3),

$$\gcd(n \cdot a, n \cdot b) = n \cdot (x^\star \cdot a = y^\star \cdot b) = n \cdot \gcd(a, b).$$

□

Theorem 7.5 (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.

Proof. We first show $M \subseteq L(a, b)$. We know that $d^\star = s^\star = x^\star \cdot a + y^\star \cdot b \in L(a, b)$. Necessarily for any $k \in \mathbb{Z}$

$$M \ni k \cdot d^\star = k \cdot (x^\star \cdot a + y^\star \cdot b) = (k \cdot x^\star) \cdot a + (k \cdot y^\star) \cdot b \in L(a, b).$$

Next we show $L(a, b) \subseteq M$. Take an arbitrary $s = x \cdot a + y \cdot b \in L(a, b)$. As $d^\star$ is a common divisor, we can write $a = k_a \cdot d^\star$ and $b = k_b \cdot d^\star$. Then

$$s = x \cdot a + y \cdot b = x \cdot (k_a \cdot d^\star) + y \cdot (k_b \cdot d^\star) = (x \cdot k_a + y \cdot k_b) \cdot d^\star \in M.$$

As $M \subseteq L(a, b)$ and $L(a, b) \subseteq M$, necessarily the two sets are the same. □

Theorem 7.6 (Reduction theorem)**.** For any $a, b \in \mathbb{Z}$,

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

Proof. Let $d_1^\star = \gcd(a, b)$ and $d_2^\star = \gcd(a - b, b)$. We show that both $d_1^\star | d_2^\star$ and $d_2^\star | d_1^\star$, then by non-negativity, $d_1^\star = d_2^\star$ follows.

Let us first show $d_1^\star | d_2^\star$. Since $d_1^\star = \gcd(a, b)$, by Theorem 7.5, it is also a divisor of all $s \in L(a, b)$, in particular, $d_1^\star | (a - b)$. As a common divisor of b and $a - b$, $d_1^\star | \gcd(a - b, b) = d_2^\star$.

Next we show $d_2^\star | d_1^\star$. Since $d_2^\star = \gcd(a - b, b)$, by Theorem 7.5, it is also a divisor of all $s \in L(a - b, b)$, in particular, $d_1^\star | ((a - b) + b) = a$. As a common divisor of a and b, $d_2^\star | \gcd(a, b) = d_1^\star$.

We have indeed shown that $d_1^\star | d_2^\star$ and $d_2^\star | d_1^\star$. As both are non-negative, necessarily $d_1^\star = d_2^\star$. □

Theorem 7.7 (Recursion of the greatest common divisor)**.** Let $n, a \in \mathbb{Z}$, then

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

Proof. Since $(a \mod b) = r = a - q \cdot b$, this follows by applying the reduction theorem repeatedly. □

7.3 Linear congruence equation

Definition 7.7 (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).

We will most often use the definition for $n \geq 2$.

Yet another way to look at this is: $a \equiv b \mod n$ exactly when $b = a + k \cdot n$ for some $k \in \mathbb{Z}$. All integers can be divided into n many congruent classes based on their remainder modulo n. Neighboring elements of these classes sit n apart on the integer line. For example,

$$\ldots \equiv -8 \equiv -3 \equiv 2 \equiv 7 \equiv 12 \equiv 17 \equiv \ldots \quad \mod 5$$

Theorem 7.8 (Operations with congruence)**.** *Let $a, b, c, d, n \in \mathbb{Z}$, $n \neq 0$, and suppose $a \equiv b \mod n$, $c \equiv d \mod n$. Then*

1. $a \pm c \equiv b \pm d \mod n$
2. $a \cdot c \equiv b \cdot d \mod n$
3. if $k \in \mathbb{Z}$ such that $k|a$, $k|b$ and $\gcd(k, n) = 1$, then $\frac{a}{k} \equiv \frac{b}{k} \mod n$.
4. for $m|n$, $a \equiv b \mod m$.

Definition 7.8 (Linear congruence equation)**.** We call the equation

$$a \cdot x \equiv b \mod n, \tag{7.4}$$

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

Most commonly, $n \geq 2$, $0 \leq a, b < n$.

Theorem 7.9 (Solvability of the linear congruence equation)**.** *Consider the linear congruence equation (7.4), 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:*

$$\begin{aligned} x_0 &= x^\star \cdot \frac{b}{d^\star} \mod n, \\ x_i &= x_0 + i \cdot \frac{n}{d^\star} \mod n \\ &= x_{i-1} + \frac{n}{d^\star} \mod n, \qquad \text{for } i = 1, \ldots, d^\star - 1. \end{aligned} \tag{7.5}$$

Proof. Suppose a solution exists. By the definition of congruence, $ax - b$ must be a multiple of n, let us write $ax - b = qn$, which we can rearrange

$$b = ax - qn.$$

That is, if a solution x exists, necessarily b is a linear combination of a and n. By Theorem 7.5, this is true exactly when b is a multiple of $d^\star$. Conversely, if b is *not* a multiple of $d^\star$, *no solution* can exist, we have proven statement 1. of the theorem.

Now consider the case when b is a multiple of $d^\star$, say, $b = k \cdot d^\star$. Let us start with $d^\star = x^\star a + y^\star n$, and consider both sides modulo n. As $y^\star n \equiv 0 \mod n$, necessarily

$$d^\star \equiv x^\star a \mod n.$$

Multiplied by k,

$$b = kd^\star \equiv kx^\star a = a(kx^\star) = a \cdot \left(x^\star \frac{b}{d^\star}\right) \mod n.$$

This exactly means that $x^\star \frac{b}{d^\star}$ is a solution, hence we have our first solution in $[0, n)$ given by

$$x_0 = \left(x^\star \frac{b}{d^\star} \mod n\right).$$

Let us look for other solutions in the form $x = x_0 + c$. Substituting into (7.4),

$$ax = a(x_0 + c) = ax_0 + ac \equiv b \mod n.$$

We already know $ax_0 \equiv b \mod n$, hence necessarily $ac \equiv 0 \mod n$. In other words, ac is a multiple of n, let us write $ac = k_2 n$ for some $k_2 \in \mathbb{Z}$. Let us simplify by $d^\star = \gcd(a, n)$, then

$$\frac{a}{d^\star} c = k_2 \frac{n}{d^\star}.$$

As $d^\star = \gcd(n, a)$, the quotients $\frac{a}{d^\star}, \frac{n}{d^\star}$ are *whole* numbers, and so are c and k_2. As the right hand side is a multiple of $\frac{n}{d^\star}$, so is the left hand side. Again, since we have simplified by $d^\star = \gcd(n, a)$, the quotients $\frac{a}{d^\star}, \frac{n}{d^\star}$ share no common factor, necessarily c is a multiple of $\frac{n}{d^\star}$. We conclude that all solutions $x = x_0 + c$ of (7.4) can be written in the form

$$x_i = x_0 + i\frac{n}{d^\star}.$$

Note that for $i = d^\star$,

$$x_{d^\star} = x_0 + d^\star \frac{n}{d^\star} = x_0 + n \equiv x_0 \mod n.$$

That is, after adding $\frac{n}{d^\star}$ (at least) $d^\star$ times, the solutions we get are congruent to solutions we have already seen. That is, the number of incongruent solutions in $[0, n)$ is $d^\star$, and we can get all of them as

$$x_i = x_0 + i\frac{n}{d^\star} \mod n, \qquad \text{for } i = 0, \ldots, d^\star - 1.$$

This concludes the proof of statement 2. of the theorem. □

As the extended Euclidean algorithm provides not only $d^\star$, but also $x^\star$, we can create an algorithm to solve the linear congruence equation.

Algorithm 7.1 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)    // We want to write the linear combination as
    $xa + yn$, hence the change in parameter names.
5:  IF $d|b$ THEN
6:      length[X] ← $d$
7:      $X_0 \leftarrow \left(x\frac{b}{d}\right) \mod n$
8:      FOR i ← 1 TO d-1 DO
9:          $X_i \leftarrow \left(x_0 + i\frac{n}{d}\right) \mod n$
10: ELSE
11:     length[X] ← 0
12: RETURN(X)
```

7.3.1 Multiplicative inverse

Definition 7.9 (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 $x_0 = a^{-1} \mod n$.

7.4 Fermat primality test and RSA cryptography

7.4.1 Modular exponentiation

For the upcoming Fermat primality test and the RSA cryptosystem, we often have to calculate modular exponentiation, that is,

$$c = (a^b \mod n).$$

It sounds quite wasteful to calculate a^b first and then calculate the remainder modulo n. We might save storage and time by calculating $a^i \mod n$ for $i = 1, \dots, b$ in order, however that is still $O(b)$ steps, and we can in fact do better than that.

We will take inspiration from the Horner method (Section 2.2.3.1). Suppose the exponent b is given in binary (this is a given when working on the computer), with bits (digits)

$$b = b_k b_{k-1} \dots b_1 b_{0(2)},$$

where $k = \lfloor \log_2(b) \rfloor$ (Theorem 2.1). Imagine we simultaneously calculate $a^b \mod n$, as we calculate the value of b in decimal with the Horner method (in the algorithm, of course we will not include the calculation of b, as we already know its value). The corresponding operations are:

- when the exponent b is multiplied by 2, a^b is squared
- when we add a new digit b_i to the exponent b, a^b is multiplied by a^{b_i} – in particular, for $b_i = 0$, a^b does not change, and for $b_i = 1$, a^b is multiplied by a.

We show the pseudocode of the algorithm first.

Algorithm 7.2 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)
```

This algorithm has time complexity $O(k) = O(\log(b))$, which clearly scales better than the naive approach with $O(b)$ operations.

We now show a table of how we calculate this by hand. For the for loop, we have columns i that counts down from k to 0, column b_i for the digit, column $c^2 \mod n$, and column $c \cdot a \mod n$ that is only filled when $b_i = 1$. We may even cross out these unused cells beforehand. The cells contain the values of c as it evolves, from left to right (when applicable) and top to bottom. We do not have a dedicated cell for initializing c as 1, but for $i = k$, we write $1^2 = 1$ in the $c^2 \mod n$ column anyways.

For our example, let us take $3^{19} \mod 16$. We first write $b = 19 = 10011_{(2)}$ and prepare our table.

i	b_i	$c^2 \bmod n$ ($c^2 \bmod 16$)	$c \cdot a \bmod n$ ($c \cdot 3 \bmod 16$)
$k = 4$	1	$1^2 \bmod 16 = 1$	$1 \cdot 3 \bmod 16 = 3$
3	0	$3^2 \bmod 16 = 9$	▨
2	0	$9^2 \bmod 16 = 81 \bmod 16 = 1$	▨
1	1	$1^2 \bmod 16 = 1$	$1 \cdot 3 \bmod 16 = 3$
0	1	$3^2 \bmod 16 = 9$	$9 \cdot 3 \bmod 16 = 27 \bmod 16 = 11$

7.4.2 Fermat primality test

For the upcoming RSA cryptosystem, one must find large prime numbers. In practice, it is hard to ensure a large number is a prime. Here, we will show a randomized primality test based on Fermat's little theorem, which cannot say with absolute certainty if a number is prime, but at least runs fast, and by repeating this test, we can increase our certainty that our number is prime. However, this is clearly not sufficient when the application hinges on whether a number is prime.

Theorem 7.10 (Fermat's little theorem)**.** If p is a prime number, then for all $a = 1, \dots, p-1$,

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

The proof of this theorem is beyond the scope of this course.
We apply the theorem as follows. Consider a number $n \in \mathbb{Z}^+$ that we want to test.

- if n is prime, the congruence must hold for all $a < n$.
- if n is not prime, in other words, it is a composite number, the congruence may or may not hold.
  - if we find $a < n$ such that $a^{n-1} \not\equiv 1 \mod n$, we can say for *certain* that n is not prime. (We call a a Fermat witness to n being a composite number.)
  - however, there is a small proportion of *Fermat pseudoprimes* to base a, where the congruence holds. (We call a a Fermat liar.)
  - the pseudoprimes are usually different numbers for different bases a. We can calculate $a^{n-1} \bmod n$ for a couple more as, if we always get 1, we have a large chance that n is a prime. We often start with $a = 2$, then check for other small as.
  - however, so called Carmichael numbers exist, that are not primes, but do satisfy $a^{n-1} \equiv 1 \mod n$ for all $a = 1, \dots, n$. There is only 255 of them among the first billion positive integers, hence the chance of being deceived by one is small, but their existence means the Fermat primality test can never definitively say a number is prime.

Let us show the psudocode for the Fermat primality test for a single base a.

Algorithm 7.3 Fermat primality test

```
1: FERMAT_TEST(n,a,@is_prime)
2: // INPUT: $a < n \in \mathbb{Z}^+$, $n$ is being tested with base $a$
3: // OUTPUT: is_prime: logical, false means for certain not prime, true means likely a prime
4: c $\leftarrow$ MOD_EXP(a,n-1,n,@c)
5: is_prime $\leftarrow$ (c = 1)
6: RETURN(is_prime)
```

7.4.3 RSA cryptosystem

The system is named after its developers: Rivest, Shamir and Adleman.

In this system, each person has two keys: a so-called private key P and a secret key S. They are a *matching* pair, inverse to each other: a message encoded with one can only be decoded by the other. The pair of keys are generated by the person themselves, and they keep their secret key secret, while the private key is shared publicly (possibly called a private key as it is a personal one that is the pair of their secret key). Messages sent to a person should be encoded by their private key, so that only they can decode it with their secret key. For digital signatures, they send their identity unencrypted as well as encoded with their secret key, so that it can be decoded with their private key and get the same content as the unencrypted identity, which verifies they indeed possess the inverse of their private key, aka their secret key.

What ensures bad actors cannot "invert" the private key to obtain the secret key? We will soon see how the keys are generated and how the encryption works. To break the encryption, one would need to factorize truly large numbers (several hundred digits). We do not have efficient algorithms for that, and with traditional digital computers, we do not have enough computational power, breaking the encryption would take several hundreds of years. However, factorization will become possible and efficient with quantum computers, the quantum algorithms already exist in theory. Once quantum computers become commonplace and powerful enough, we will have to switch over to other cryptosystems.

How the RSA keys are generated:

1. first, two large primes, p and q are chosen (in real-world applications, these can be 100-200 digits long).
2. we calculate $n = p \cdot q$ and $f = (p-1) \cdot (q-1)$.
3. we choose a small odd number $e \geq 3$ such that $\gcd(e, f) = 1$.
4. we calculate $d = e^{-1} \mod f$.
5. the private key is $P = (n, e)$. A message M is encrypted as $M^e \mod n = C$.
6. the secret key is $S = (n, d)$. An encoded message C is decrypted as $C^d \mod n = M$.
7. The message should be $M \in \mathbb{N}, M < n$. We can achieve this for example by splitting the bit sequence into chunks and reading those chunks as binary numbers (unsigned integers), which will be encrypted separately.

Let us give an idea why the decoding works, this will not be a proper proof. Let the coded message

$$C = M^e \mod n.$$

and the coded and then decoded message

$$M' = C^d \mod n.$$

We put these together

$$M' = (M^e)^d \mod n = M^{e \cdot d} \mod n.$$

Recall that d was chosen as $d = e^{-1} \mod f$, which means $d \cdot e \equiv 1 \mod f$, or in other words, $d \cdot e = k \cdot f + 1$ for some $k \in \mathbb{Z}$. Plugging this in,

$$M' = M^{kf+1} \mod n = (M^f)^k \cdot M \mod n.$$

By the little Fermat theorem, for all $a < p$ and respectively, $a < q$

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

This is the not precise part, but it “looks right” to then say,

$$a^{(p-1)(q-1)} \equiv 1 \mod (p \cdot q).$$

Here, we recognize $f = (p-1)(q-1)$ and $n = pq$. We once again take a leap of faith by taking $a = M$, then we get

$$M^f \equiv 1 \mod n.$$

Once again plugging this in,

$$M' = (M^f)^k \cdot M \mod n = 1^k \cdot M \mod n = M \mod n,$$

as we wanted.

We now write the generation of RSA keys as an algorithm.

Algorithm 7.4 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 ← NULL, S ← NULL, 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)
```

Chapter 8

Instruction

Back to top

Contents of chapter

8.1 The greatest common divisor

8.1.1 Basic algorithms for the greatest common divisor

8.1.1.1 Prime factorization

This method is based on finding the prime factorization of a and b separately. For example,

$$\begin{array}{r|l} 180 & 2 \\ 90 & 2 \\ 45 & 3 \\ 15 & 3 \\ 5 & 5 \\ 1 & \end{array} \quad 180 = 2^2 \cdot 3^2 \cdot 5 \qquad\qquad \begin{array}{r|l} 144 & 2 \\ 72 & 2 \\ 36 & 2 \\ 18 & 2 \\ 9 & 3 \\ 3 & 3 \\ 1 & \end{array} \quad 144 = 2^4 \cdot 3^2$$

Then

$$\gcd(180, 144) = 2^{\min\{2,4\}} \cdot 3^{\min\{2,2\}} \cdot 5^{\min\{0,1\}} = 2^2 \cdot 3^2 \cdot 5^0 = 36.$$

This algorithm is not efficient for large numbers, because finding primes by itself is not efficient.

8.1.1.2 Naive method using the reduction theorem

We make use of the following properties:

- $\gcd(a, 0) = a$

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

And construct the following algorithm:

Algorithm 8.1 Naive GCD based on the reduction theorem

```
NAIVE_GCD(a,b,@d)
// INPUT: $a, b, \in \mathbb{N}, a \geq b$
// OUTPUT: $d = \gcd(a, b) \in \mathbb{N}$
WHILE $b > 0$ DO
    IF $b > a$ THEN
        swap $a \leftrightarrow b$      // in most programming languages, this means $c \leftarrow a$, $a \leftarrow b$, $b \leftarrow c$
    ELSE
        $a \leftarrow a - b$
$d \leftarrow a$
RETURN(d)
```

Noting it down by hand, this might look like:

$$\begin{aligned}\gcd(180, 144) &= \gcd(36, 144) = \gcd(144, 36) = \gcd(108, 36) = \gcd(72, 36)\\ &= \gcd(36, 36) = \gcd(0, 36) = \gcd(36, 0) = 36.\end{aligned}$$

This scales better than prime factorization, but still often ends up being very slow, since for small values of b, $a - b$ will be a small decrease compared to a.

8.1.1.3 Binary algorithm for the greatest common divisor

The binary algorithm combines some of the ideas seen above, while also relying on operations that are efficient when a and b are represented as binary (base two) numbers on the computer.

We rely on the following observations, which will become the possible steps of the algorithm:

- $\gcd(a, 0) = a$, this is our clue the algorithm is finished.
- $\gcd(a, b) = \gcd(b, a)$, we use this step to make sure $a \geq b$ throughout.
- as a special case of Corollary 7.2, that is, the property that

  $$\gcd(n \cdot a, n \cdot b) = n \cdot \gcd(a, b),$$

  we observe that when a and b are both even, we can separate a factor 2, and quickly reduce the numbers. (This is also efficient, as dividing a binary number by two is simply a digit shift.)
- we also note that, when one number is even and the other is not, we cannot separate the factor 2 (recall the GCD by prime factorisation), hence we might as well divide the even number by 2 to quickly reduce it.
- when both numbers are odd, we rely on the reduction theorem and replace a by $a - b$. However we also combine this with the step above, knowing b is odd while $a - b$ is even, hence we take $\frac{a-b}{2}$ instead.

Let us summarize in formula.

$$\gcd(a,b) = \begin{cases} a, & \text{if } b = 0, \\ \gcd(b,a), & \text{if } b > a, \\ 2 \cdot \gcd(a/2, b/2), & \text{if } a \text{ and } b \text{ are both even,} \\ \gcd(a/2, b), & \text{if } a \text{ is even and } b \text{ is odd,} \\ \gcd(a, b/2), & \text{if } a \text{ is odd and } b \text{ is even,} \\ \gcd\big(a, (a-b)/2\big), & \text{if } a \text{ and } b \text{ are both odd.} \end{cases}$$

And also show the pseudocode for the algorithm that implements these ideas.

Algorithm 8.2 Binary algorithm for GCD

```
BINARY_GCD(a,b,@d)
// INPUT: $a, b \in \mathbb{N}, a \geq b$
// OUTPUT: $d = \gcd(a,b) \in \mathbb{N}$
$c \leftarrow 1$        // this variable will store the separated 2 factors
WHILE $b > 0$ DO
    $r_a \leftarrow (a \bmod 2)$
    $r_b \leftarrow (b \bmod 2)$        // these show parity: 0 is even, 1 is odd
    CASES
      CASE $r_a = 0$, $r_b = 0$:
          $c \leftarrow 2c$
          $a \leftarrow a/2$
          $b \leftarrow b/2$
      CASE $r_a = 0$, $r_b = 1$:
          $a \leftarrow a/2$
      CASE $r_a = 1$, $r_b = 0$:
          $b \leftarrow b/2$
      CASE $r_a = 1$, $r_b = 1$:
          $a \leftarrow \frac{a-b}{2}$
    IF $a < b$ THEN
        swap $a \leftrightarrow b$
$d \leftarrow c \cdot a$
RETURN(d)
```

Writing it by hand, if may look like

$$\begin{aligned} \gcd(180,144) &= 2 \cdot \gcd(90,72) = 4 \cdot \gcd(45,36) = 4 \cdot \gcd(45,18) = 4 \cdot \gcd(45,9) \\ &= 4 \cdot \gcd\big(\underbrace{(45-9)/2}_{18}, 9\big) = 4 \cdot \gcd(9,9) = 4 \cdot \gcd\big(\underbrace{(9-9)/2}_{0}, 9\big) \\ &= 4 \cdot \gcd(9,0) = 4 \cdot 9 = 36. \end{aligned}$$

8.1.2 The Euclidean algorithm

The Euclidean algorithm is based on the Recursion theorem, Theorem 7.7.

Repeatedly applying $\gcd(a,b) = \gcd(b, a \bmod b)$ until b becomes 0 yields the greatest common divisor *quickly*. We show it implemented as both an iterative and a recursive algorithm.

Algorithm 8.3 Euclidean algorithm, iterative version

```
1: EUCL_IT(a,b,@d)
2: // INPUT: $a, b \in \mathbb{N}, a \geq b$
3: // OUTPUT: $d = \gcd(a, b) \in \mathbb{N}$
4: WHILE $b > 0$ DO
5:     $r \leftarrow b \bmod a$
6:     $a \leftarrow b$
7:     $b \leftarrow r$
8: $d \leftarrow a$
9: RETURN(d)
```

Algorithm 8.4 Euclidean algorithm, recursive version

```
1: EUCL_IT(a,b,@d)
2: // INPUT: $a, b \in \mathbb{N}, a \geq b$
3: // OUTPUT: $d = \gcd(a, b) \in \mathbb{N}$
4: IF $b > 0$ THEN
5:     $d \leftarrow$ EUCL_REC(b,$a \bmod b$,@d)
6: ELSE
7:     $d \leftarrow a$
8: RETURN(d)
```

When we calculate it by hand, we make a table. In each row, we calculate the quotient $q_i = (a_i \text{ div } b_i)$ and remainder $r_i = (a_i \bmod b_i)$ of the whole division. When we move to the next row, our new a_{i+1} is the old b_i, and our new b_{i+1} is the old r_i. (For now, q_i is unused, but we will use it in the upcoming extended Euclidean algorithm.) We stop when $b_k = 0$, we obviously cannot divide anymore, and a_k of that row is $d^\star = \gcd(a, b)$ (or to be very precise, $\gcd(a_0, b_0)$).

index	a	b	q	r
0	210	144	1	66
1	144	66	2	12
2	66	12	5	6
3	12	6	2	0
4	6	0		

$\gcd(210, 144) = 6$

8.1.2.1 Runtime

Intuitively, this algorithm seems faster than the previous ones, as it decreases a and b much faster. To say something more precise about the runtime, we have the following theorem.

Theorem 8.1 (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 4.10). Then the number of recursive calls is less than k.

The proof of the theorem is beyond the scope of this course.

As a consequence of this theorem, knowing that the Fibonacci numbers grow exponentially (Thm 4.3), this means the number of recursions is $O(\log(b))$. More precisely, from

$$b \approx F_k \approx \frac{1}{\sqrt{5}}\Phi^k,$$

taking logarithm,

$$\log_{10}(b) \approx k \cdot \log_{10}(\Phi) - \frac{1}{2}\log_{10}(5).$$

Rearranging,

$$k \approx \frac{\log_{10}(b) + \frac{1}{2}\log_{10}(5)}{\log_{10}(\Phi)} \approx \frac{\log_{10}(b)}{\log_{10}(\Phi)}.$$

With $\Phi = \frac{1+\sqrt{5}}{2} \approx 1.618$, we have $\frac{1}{\log_{10}(\Phi)} \approx 5$, thus the number of recursions

$$k \approx 5 \cdot \log_{10}(b).$$

Also note that by Theorem 2.1, $\log_{10}(b)$ is roughly the number of digits of b (in base ten).

The above number shows the number of divisions required. If we take into account that the number of processor operations required to calculate such a division is proportional to the number of digits, we in fact get a runtime of $O(\log(a) \cdot \log(b))$.

8.1.2.2 The extended Euclidean algorithm

In some upcoming applications, it will be useful to not only calculate the greatest common divisor $d^\star$ of $a, b \in \mathbb{Z}$, but also the coefficients $x^\star, y^\star \in \mathbb{Z}$ of the linear combination $d^\star = x^\star \cdot a + y^\star \cdot b$.

We do so by extending the table of the Euclidean algorithm we have prepared for calculations by hand with two additional columns x and y. Our aim is to determine x_i and y_i such that for each row i, we have $x_i \cdot a_i + y_i \cdot b_i = d^\star$. (This is possible, since by the recursion theorem, $\gcd(a_i, b_i) = d^\star$ in all rows, and Theorem 7.5 guarantees $d^\star$ can be written as a linear combination.)

Then the first row $i = 0$ will have our desired result $d^\star = x_0 \cdot a_0 + y_0 \cdot b_0 = x^\star \cdot a + y^\star \cdot b$. For the last row, with $a_k = d^\star$ and $b_k = 0$, we have a trivial choice of $x_k = 1$ and $y_k = 0$. We will then fill the columns x and y from *bottom to top*, using the recursion

$$x_{i-1} = y_i, \qquad y_{i-1} = x_i - y_i \cdot q_{i-1}.$$

To show the recursion in action, let us extend the table of our previous example:

index	a	b	q	r	x	y
0	210	144	1	66	11	-5 - 11 · 1 = -16
1	144	66	2	12	-5	1 - (-5) · 2 = 11
2	66	12	5	6	1	0 - 1 · 5 = -5
3	12	6	2	0	0	1 - 0 · 2 = 1
4	6	0			1	0

From row $i = 0$, we read $x^\star = x_0 = 11$ and $y^\star = y_0 = -16$. To double check, we calculate

$$x^\star \cdot n + y^\star \cdot a = 11 \cdot 210 - 16 \cdot 144 = 2310 - 2304 = 6,$$

which is indeed $d^\star$.

Let us now show how the recursion arises. What we want to achieve is

$$d^\star = x_i \cdot a_i + y_i \cdot b_i = x_{i-1} \cdot a_{i-1} + y_{i-1} \cdot b_{i-1}. \tag{8.1}$$

When we filled out the first half of the table, we calculated the whole division

$$a_{i-1} = q_{i-1} \cdot b_{i-1} + r_{i-1},$$

and moved on to the next row with

$$a_i = b_{i-1}, \qquad b_i = r_{i-1}.$$

Substituting this into the whole division,

$$a_{i-1} = q_{i-1} \cdot a_i + b_i.$$

Let us substitute all these on the right hand side of (8.1):

$$\begin{aligned} x_i \cdot a_i + y_i \cdot b_i &= x_{i-1} \cdot a_{i-1} + y_{i-1} \cdot b_{i-1} \\ &= x_{i-1} \cdot (q_{i-1} \cdot a_i + b_i) + y_{i-1} \cdot a_i \\ &= (x_{i-1} \cdot q_{i-1} + y_{i-1}) \cdot a_i + x_{i-1} \cdot b_i \end{aligned}$$

We note that both the left hand side and the right hand side are linear combinations of a_i and b_i. We can make sure they are equal by making the coefficients equal:

$$x_i = x_{i-1} \cdot q_{i-1} + y_{i-1}, \qquad y_i = x_{i-1}.$$

Rearranging, we indeed get

$$x_{i-1} = y_i, \qquad y_{i-1} = x_i - x_{i-1} \cdot q_{i-1} = x_i - y_i \cdot q_{i-1}.$$

When we calculate by hand, we first prepare the table of the Euclidean algorithm as usual, from top to bottom, then add the columns of x and y, that we fill from bottom to top. Let us now examine how we can implement the extended Euclidean algorithm as a program, whether we can suitably modify Algorithms 8.3 and 8.4.

To extend the iterative Algorithm 8.3, we would have to store at least parts of the table, as the q_i are used in reverse order. We do not pursue this method further.

However, luckily for us, the recursive Algorithm 8.4 can be adapted to calculate x_i and y_i when we ascend back on the recursive call tree. Rather than simply returning $d^\star$, we return $d^\star$, x_i and y_i. When we receive these variables from the recursive call, we receive x_i and y_i from the row below. Then we can calculate the new x_{i-1} and y_{i-1}, and return these alongside $d^\star$. For the basic case $b = 0$, we initialize $x = 1$, $y = 0$.

Algorithm 8.5 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: $d = \gcd(a, b)$
4: // OUTPUT: $x, y \in \mathbb{Z}$ coefficients of linear combination $d = x \cdot a + y \cdot b$
5: IF $b > 0$ THEN
6:     $q \leftarrow \left\lfloor \frac{a}{b} \right\rfloor$
7:     $r \leftarrow a - b \cdot q$
8:     (d,x_old,y_old) ← EXT_EUCL_REC(b,r,@d,@x,@y)
9:     x ← y_old
10:    y ← x_old - y_old · q
11: ELSE
12:    d ← a
13:    x ← 1
14:    y ← 0
15: **RETURN**(d,x,y)
```

Chapter 9

Exercises

Back to top

Contents of chapter

9.1 Greatest common divisor

Exercise 9.1 (Greatest common divisor)**.** Find the greatest common divisor of $a = 84$ and $b = 72$ using:

a) prime factorization,

b) reduction theorem (naive method),

c) binary algorithm.

Exercise 9.2 (Extended Euclidean algorithm)**.** Use the extended Euclidean algorithm to find the greatest common divisor $d = \gcd(a, b)$, as well as the coefficients $x, y \in \mathbb{Z}$ to write $d = xa + yb$, for numbers:

a) $a = 960$, $b = 102$

b) $a = 975$, $b = 600$

c) $a = 208$, $b = 101$

d) $a = 55$, $b = 34$

9.2 Linear congruence equation

Exercise 9.3 (Linear congruence equation). Solve the following linear congruence equations:

a) $5x \equiv 8 \mod 12$

b) $3x \equiv 8 \mod 12$

c) $3x \equiv 6 \mod 12$

d) $2x \equiv 8 \mod 20$

e) $8x \equiv 2 \mod 20$

9.2.1 Multiplicative inverse

Exercise 9.4 (Multiplicative inverse). Find the multiplicative inverse for:

a) $1^{-1} \mod 10$

b) $2^{-1} \mod 10$

c) $3^{-1} \mod 10$

d) $3^{-1} \mod 20$

e) $2^{-1} \mod 7$

9.3 RSA cryptography and Fermat primality test

9.3.1 Modular exponentiation

Exercise 9.5 (Modular exponentiation). Calculate the following:

a) $5^{13} \mod 12$

b) $2^{28} \mod 10$

9.3.2 Fermat prime test

Exercise 9.6 (Fermat prime test). Test if whether base a is a Fermat witness to n being a composite number, for:

a) $a = 2$, $n = 12$

b) $a = 2$, $n = 13$

9.3.3 RSA encoding and decoding

Exercise 9.7 (RSA encoding and decoding).

a) Generate the pair of RSA keys with $p = 3$, $q = 11$, and $e = 7$; that is, calculate n, f and d.

b) Encode $M = 17$ (using the private key).

c) Decode the previous result (using the secret key) and check if you got back 17.

Part IV

Dynamic sets: array, linked list, hash table

Chapter 10

Lecture: sequence

Back to top

Contents of chapter

10.1 Dynamic sets: introduction

Definition 10.1 (Dynamic set)**.** A dynamic set is a dataset that changes (elements are added, removed, modified) during the run of the algorithm using it.

Elements of such dynamic sets may be complex data types themselves, that contain not only one data or informational field, but possibly several of those, and additional metadata fields, such as a *key*, or *pointer(s)* to other element(s) of the dynamic set.

- A *key* is an (often unique) identifier of entries in a set, most often a number. We assume the operations *less than, greater than, equal to* are defined on the set of keys.
- A *pointer* is a memory address where some data is stored. It may take the special value NIL, which means it is pointing to nowhere (not a valid memory address).

Depending on intended usage and implementation, a dynamic set may support some of the following common operations. Some of these may be missing, others may be adapted for more specific usage, or additional operations may be introduced.

In the following,

- S denotes the dynamic set,
- k denotes a key,
- and x and y denote pointers to elements of S.

Query operations

- SEARCH(S,k,@x): finds an element in S with key k and returns its pointer x (NIL if no such element is found).
- MINIMUM(S,@x): finds the smallest key in S and returns the pointer x of that element.
- MAXIMUM(S,@x): finds the largest key in S and returns the pointer x of that element.
- NEXT(S,x,@y): returns the pointer y of the element that follows x in S (NIL if no such element exists).
- PREVIOUS(S,x,@y): returns the pointer y of the element that precedes x in S (NIL if no such element exists).

Modifying operations

- INSERT(S,x): adds the element with pointer x to the set S.
- DELETE(S,x): removes the element with pointer x form the set S.

Operations may be classified as

- *static*, if they do not change the dynamic set.
- *dynamic*, if they change the dynamic set.
- note: *typically*, query operations are static, and modifying operations are dynamic. However, we can imagine a dynamic search that rearranges the order of elements.

10.1.1 Sequence

Definition 10.2 (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.

Next, we show two common implementations of the sequence: array and linked list.

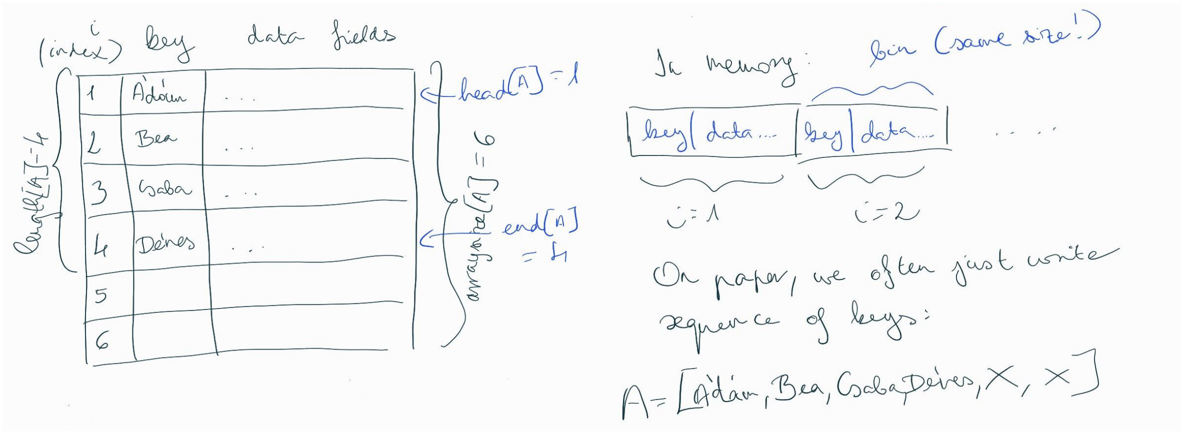
Afterwards, we will look at more complex data structures based on the sequence: queue and stack. These prescribe where an element can be inserted and which is the next element to be removed – thus, they are often used to store data that is yet to be processed by the algorithm, so we can process it in a specific order.

10.2 Array

The array implements the sequence by storing items consecutively in memory. We think of each item as a data record, e.g., all data associated with an employee (such as name, sex, age, start of employment, etc.). It is predefined which field has what data type and how much memory space it takes up. Think of the `struct` type in C. As such, each item can be stored in identically "partitioned" or structured *bins* of memory.

In this course, we do not concern ourselves with the specific data fields, we access entire records by index, or by key, for searching and sorting purposes. Typically, we will denote the array by A, an index by x (we will index arrays starting from 1, using mathematics convention), the record indexed by x as A_x, and the key field as key$[A_x]$ (other fields can be accessed analogously, by field name). For a visual representation of the array and its upcoming attributes (see below), see Figure 10.1.

Figure 10.1: Visual representations of an array, its storage method and its attributes



10.2.1 Attributes

Data structures often have *attributes*, i.e., properties or descriptors; we can often think of them as meta-data.

The array has the following attributes:

- arraysize[A]: the size of physical memory reserved for the array, given in the (whole) number of bins available for records/elements.

!! The array stores its elements in *consecutive* bins, starting from the *first bin* of the array, i.e., at the start of the allocated memory.

!! We index elements starting from 1.

- length[A]: the number of (currently) stored elements, 0 if there are no elements. Must always satisfy length[A] $\leq$ arraysize[A].
- head[A]: the index of the first element, equals 1.
- end[A]: the index of the last element, 0 if there are no elements, equals length[A].

Let us emphasize and spell out what the attributes already suggest.

On arraysize and length: As the array stores elements consecutively in the memory, a continuous chunk of memory must be reserved in advance. We usually measure this in the amount of bins available for data records. This is the arraysize, which is an upper limit for how many records we can store at once. We may store less, which allows the array to function as a dynamic storage, as its length can change. Clearly, the arraysize must be chosen as a smart compromise. If we run out of available bins, often the only way to increase the storage size is to reserve a bigger chunk of memory elsewhere and move the whole array, which is costly. However, we also do not want to occupy too much memory unnecessarily.

First and last index: We only keep the first and last occupied index as attributes, and do not keep individual flags whether each bin is in use or not[1], that is, whether the bits there are our stored data or just memory junk.[2] As a consequence, we make sure the bins we use are consecutive ones, starting from the first one in the array. We also have to make sure to maintain this property during the upcoming supported/implemented operations.

10.2.2 Generalizations

An array is sometimes called a vector, which emphasizes its one-dimensional nature. It can also be generalized to support multiple indices, for example an object with two indices is often called a table or matrix, where the first index gives the row and the second index gives the column. We can think of such an object as a vector that contains vectors. Keep in mind that the computer's memory is a linear sequence of bins, hence conventions must be created to decide the storage order of elements of a two- or more-dimensional array (can also think of this as a function that maps the two-dimensional index pair of the matrix into the one-dimensional single index of the underlying 1-dimensional storage array). A common convention is to store a matrix row-wise: we store rows of the matrix consecutively. Analogously, we can store it column-wise.

10.2.3 Operations

The array has operations search, insert, and delete. We introduce their algorithms below.

10.2.3.1 Insert

We allow inserting a new record into the array at any index (more precisely, currently within bounds or at the end). As records are stored consecutively, space can only be created by moving each record after the desired index into the bin after its current position. Worst case scenario, we insert at index 1, and all previous records must be moved, thus this operation has runtime O(length[A]). The pseudocode is in Algorithm 10.1. An example run of a successful insertion is shown in Figure 10.1.

10.2.3.2 Delete

We remove a record by its index (if it is an existing index). To remove a record by its key, we can simply combine a search and a delete operation. Once again, so that records are stored consecutively, all records after the deleted one must be moved one bin earlier. In the worst case scenario, if the first record is deleted, all other records must be moved, hence this operation also has runtime O(length[A]). The pseudocode is in Algorithm 10.2. An example run of a successful deletion is shown in Figure 10.2.

10.2.3.3 Search

The aim of the search operation is to find the (first occurrence) of key k in the array A, and return the pointer to the corresponding record A_x (or its index x), or NIL if no such record exists. We introduce two variants. Without assuming the elements of the array are ordered, the best we can do is *linear*

[1]That would also be a valid approach, but result in a different data structure from than the array we discuss here.

[2]Remember, bits cannot be "empty", they are always 0s or 1s, and we can always read them according to our data type, but it may be meaningless to us if it's been left there from earlier, that is what we refer to as memory junk.

Algorithm 10.1 Insertion into array

```
1: INSERT(A,r,x,@message)
2: // INPUT: A array, r record to be inserted, x index where it should be inserted    // r is inserted before current $A_x$
3: // OUTPUT: message: success or error message
4: IF length[A] $\geq$ arraysize[A] THEN
5:     message ← "array is full"
6: ELSE IF x > end[A] + 1 THEN
7:     message ← "index out of bounds"
8: ELSE    // Otherwise we can insert; attaching at the end of the array, x = end[A]+1 is also included in this case.
9:     FOR i ← end[A] DOWNTO x DO    // We assume end[A] < x results in the loop body never being executed.
10:        $A_{i+1} \leftarrow A_i$    // Move each element into the next bin. From end to middle so none gets overwritten.
11:    $A_x \leftarrow$ r
12:    INC(length[A])
13:    INC(end[A])
14:    message ← "successfully inserted"
15: RETURN(message)
```

search. When we know the elements of the array are ordered, we can take advantage of this, and use the more efficient *binary search.*

Linear search: Let the index run from 1, look at the corresponding record and compare its key to k. We increase the index until we either find a key equal to k, or reach the end of the array. The pseudocode is in Algorithm 10.3. In the worst case scenario, when k is not found, we must look at each record, hence this algorithm has runtime O(length[A]). Example runs of a successful and an unsuccessful linear search are shown in Figure 10.3.

Binary search: We can improve on the above linear search, if we know the array is already *sorted.* Throughout this course, by a sorted sequence, we will mean *sorted in increasing order of the key field.*

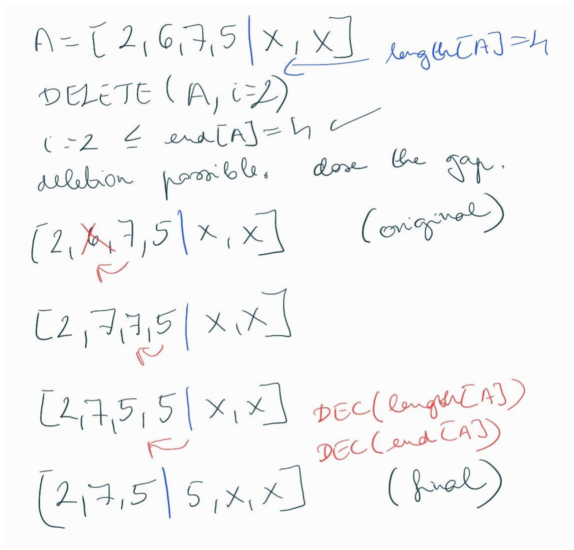
The idea is to first check the middle of the sequence, and compare the key field of the record found there to the key we are searching for. If they are equal, we are lucky. If not, we know if we have to continue our search among smaller keys, that is, in the first half of the array, or among larger keys in the second half of the array. We continue by again checking the middle element of the sub-sequence first, and so on. The pseudocode is given in two versions: Algorithm 10.4 shows an iterative approach, and Algorithm 10.5 shows a recursive approach. As we can halve the array $\lceil \log_2(\text{length[A]}) \rceil$ times until we are left with a 1-element sequence, the runtime of both of these algorithms is $O\big(\log_2(\text{length[A]})\big)$. Figure 10.4 shows example runs of a successful and an unsuccessful binary search.

Algorithm 10.2 Deletion from array (by index)

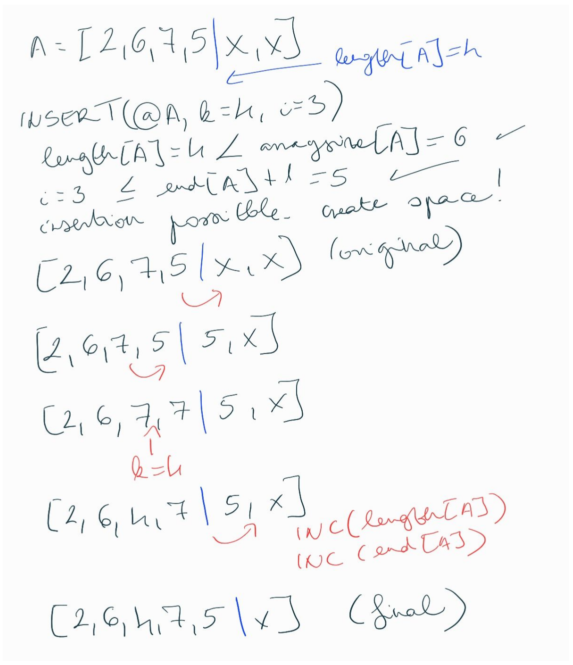
```
1: DELETE(A,x,@message)
2: // INPUT: A array, x index of $A_x$ record to be deleted
3: // OUTPUT: message: success or error message
4: IF length[A] = 0 THEN
5:     message ← "empty array"
6: ELSE IF x > end[A] THEN
7:     message ← "index out of bounds"
8: ELSE
9:     FOR i ← x TO end[A]-1 DO      // Assume that x > end[A]-1 means an "empty loop", that is, the loop body is never executed.
10:        $A_i$ ← $A_{i+1}$      // Move each element into the previous bin. From middle to end, so none gets overwritten.
11:    DEC(length[A])
12:    DEC(end[A])
13:    message ← "successfully deleted"
14: RETURN(message)
```

Figure 10.2: Modifying operations of an array

(a) Deletion from array



(b) Insertion into array

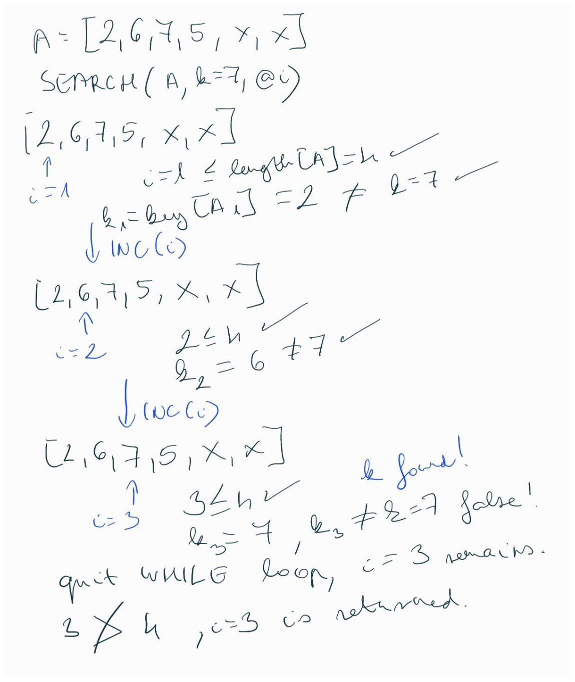


Algorithm 10.3 Linear search in array

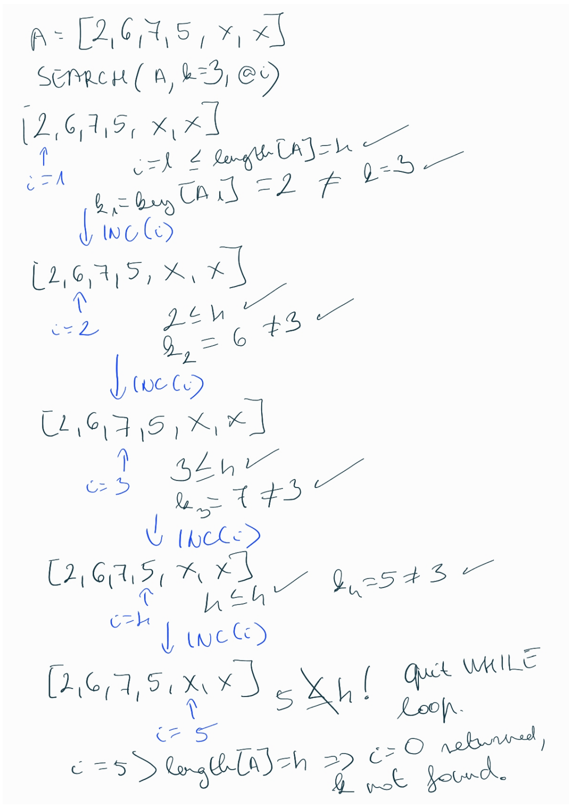
```
LIN_SEARCH(A,k,@x)
// INPUT: A array, k key to be found
// OUTPUT: x: index of a record that key[$A_x$] = k, or 0 if no such record exists
IF length[A] = 0 THEN
    x ← 0
ELSE
    x ← head[A]
    WHILE x ≤ end[A] AND key[$A_x$] ≠ k DO    // For this to not give “out of bound” type errors, we assume the operation AND is implemented efficiently, that is, if the first condition fails, AND is already false, and the second condition is not even checked.
        INC(x)
    IF x > end[A] THEN    // The while loop stopped either because we have found x such that key[$A_x$] = k, or because x > end[A].
        x ← 0
RETURN(x)
```

Figure 10.3: Linear search in an array

(a) Successful



(b) Unsuccessful



Algorithm 10.4 Binary search in *ordered* array, iterative version

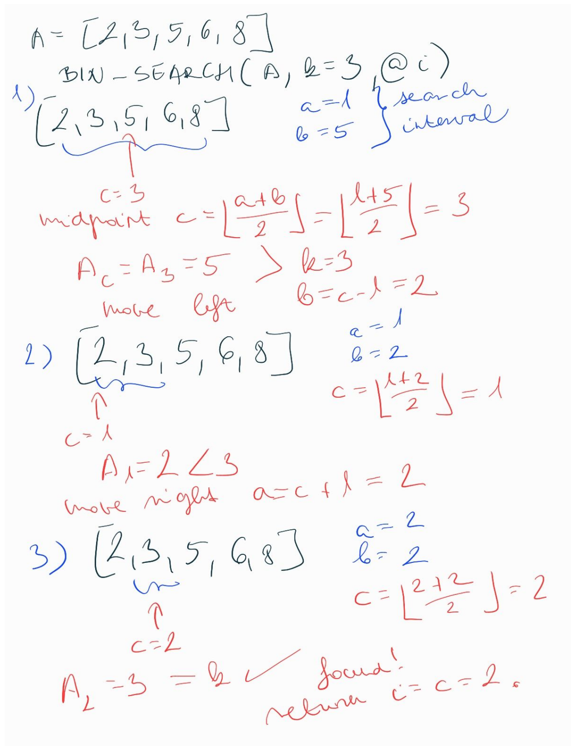
```
1: BIN_SEARCH_IT(A,k,@x)
2: // INPUT: array A, key k to search
3: // OUTPUT: x index so that key[$A_x$]=k or 0 if no such record is found
4: IF length[A] = 0 THEN
5:     x ← 0
6:     RETURN(x)
7: a ← head[A]
8: b ← end[A]      // These are the first and last index of the subsequence we are still searching.
9: WHILE a ≤ b DO
10:     c ← $\lfloor \frac{a+b}{2} \rfloor$      // 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 ← $c-1$      // search in first half
16:     ELSE      // necessarily key[$A_c$] < k
17:         a ← $c+1$      // search in second half
18: x ← 0      // If we did not return yet, k was not found.
19: RETURN(x)
```

Algorithm 10.5 Binary search in *ordered* array, recursive version

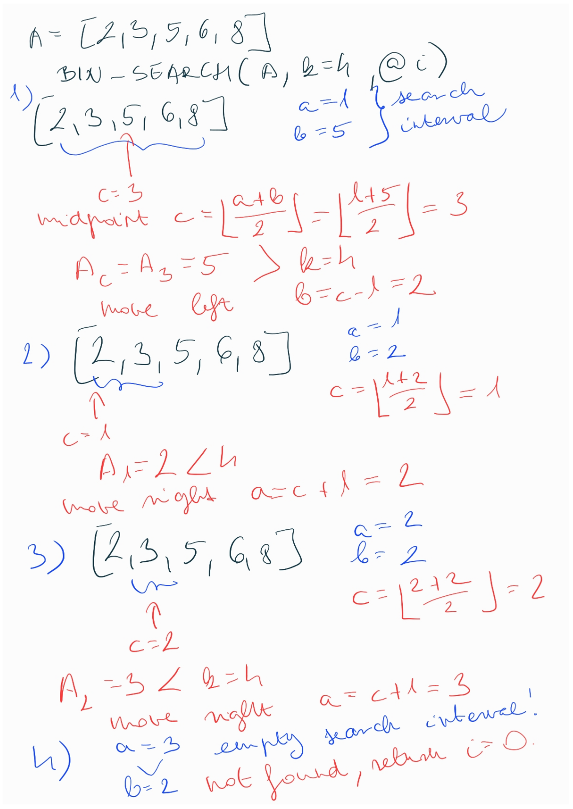
```
1: BIN_SEARCH_REC(A,k,a,b,@x)
2: // INPUT: array A, key k to search, 1 ≤ a ≤ b ≤ length[A] the index range (subsequence) we
   search      // Search whole array with a = head[A], b = end[A]
3: // OUTPUT: x index so that key[$A_x$]=k or 0 if no such record is found
4: IF a > b THEN      // this means empty subsequence
5:     x ← 0
6:     RETURN(x)
7: c ← $\lfloor \frac{a+b}{2} \rfloor$      // calculate midpoint
8: IF key[$A_c$] = k THEN
9:     x ← c
10:     RETURN(x)
11: ELSE IF k < key[$A_c$] THEN
12:     x ← BIN_SEARCH_REC(A,k,a,c-1)      // recursive call for first half
13: ELSE      // necessarily key[$A_c$] < k
14:     x ← BIN_SEARCH_REC(A,k,c+1,b)      // recursive call for second half
15: RETURN(x)
```

Figure 10.4: Binary search in an array

(a) Successful



(b) Unsuccessful



10.2.3.4 Supplementary material: other implementations of the operations

An improved linear search We can still try to improve the search algorithm, even when the array is not ordered. In Algorithm 10.3, the while loop, that causes the linear runtime, checks two conditions. We could roughly halve the runtime by dropping one condition. We achieve that by attaching a new fictive record after the end of the array with the same key k that we are searching for. Then the while loop can stop checking whether the index is still within range, as it is now guaranteed that k *will* be found, at index x = end[A]+1 at the latest. Now it is sufficient to check whether we have stopped at an index x $\leq$ end[A] and found a *real* instance of key k, or only our fictive record at x = end[A]+1. This idea is realized in Algorithm 10.6.

Algorithm 10.6 Linear search in array – improved constant

```
 1: LIN_SEARCH_IMPROVED(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] < arraysize[A] THEN     // Insert fictive record if possible. Note that we do not in-
    crease length[A] – hence this fictive record will just be considered memory junk by other algorithms
    and operations.
 5:     key[$A_{\text{end[A]}} + 1$] ← k
 6:     x ← head[A]
 7:     WHILE key[$A_x$] ≠ k DO
 8:         INC(x)
 9:     IF x > end[A] THEN     // only fictive record has key k
10:         x ← 0
11: ELSE
12:     x ← LIN_SEARCH(A,k)     // Otherwise fall back on usual linear search
13: RETURN(x)
```

Fibonacci search Another logarithmic search is based on the Fibonacci numbers. This is more so for curiosity's sake, binary search is much simpler and very effective. The Fibonacci search, instead of halving the sequence, splits it into chunks as large as two consecutive Fibonacci numbers, that is, nearly as a golden ratio.

We assume array A is ordered, and we have length[A] = n so that n+1 = F_{k+1} for some Fibonacci number. The pseudocode is given in Algorithm 10.7.

Algorithm 10.7 Fibonacci search

```
FIB_SEARCH(A,k,@i)
// INPUT: ordered array A, key k to search for
// OUTPUT: index i so that key[$A_i$] = k, or 0 if no such element exists
    // Denote n = length[A], and for simplicity, assume n+1 = $F_{k+1}$ for some Fibonacci number
i ← $F_k$
p ← $F_{k-1}$
q ← $F_{k-2}$
WHILE TRUE DO
    IF key[$A_i$] = k THEN    // k is found
        RETURN(i)
    ELSE IF key[$A_i$] < k THEN
        IF q = 0 THEN    // k is not found
            i ← 0
            RETURN(i)
        ELSE    // decrease i
            i ← i-q
            (p,q) ← (q,p-q)
    ELSE    // key[$A_i$] > k
        IF p = 1 THEN    // k is not found
            i ← 0
            RETURN(i)
        ELSE    // increase i
            i ← i+q
            p ← p-q
            q ← q-p    // With the new reduced p.
RETURN(0)
```

10.3 Linked list

Similarly to the array, data records in a linked list are stored in identically structured bins in the memory. In the more common implementation with pointers, these bins are allocated one by one as needed, hence not (necessarily) stored consecutively in memory. (Later we will talk about an array implementation as well.) We are able to follow the sequence because each element stores a *next* field with a pointer to the next element (also called successor). If an element x has no successor, next[x] = NIL. This is why we call this structure a *linked* list, as records are like links in a chain, they stand on their own, but each leads to the next. Imagine this as a treasure hunt.

Notation, attributes

We often denote a linked list by L. The list has a single attribute head[L] with a pointer to the first element. An empty list has head[L] = NIL.

Doubly linked list

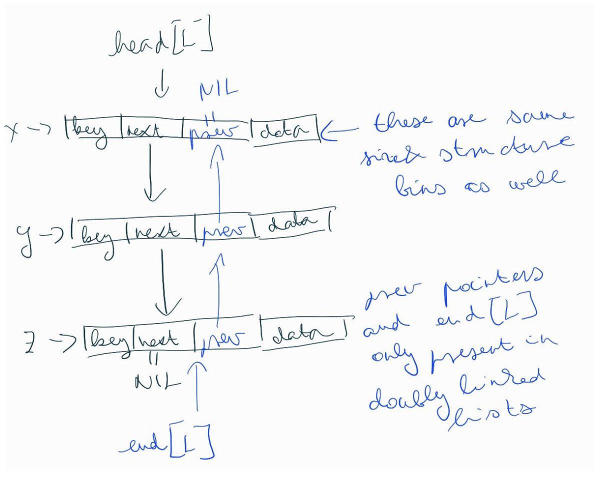
If the records store an additional *previous* field with a pointer to the previous element, we are talking about a *doubly linked list*, which is most often the case. Clearly, the first element x has prev[x] = NIL. A doubly linked list also has an end[L] attribute, that is a pointer to the last element, or again NIL if the list is empty. A visual representation of a doubly linked list is shown in Figure 10.5.

Pros and cons of the linked list VS the array

The pro of this approach is its flexibility: we do not have to estimate in advance how many records are needed, the limit is the capacity of the memory. Insertion and deletion can also become faster, as elements do not have to be moved from bin to bin, only the pointers of the neighbors must be changed.

The downside is that we cannot access elements directly by index, as we only have the pointer to

Figure 10.5: Linked list


head[L]
NIL
x → key next prev data
these are same sized structure bins as well
y → key next prev data
z → key next prev data
NIL
prev pointers and end[L] only present in doubly linked lists
end[L]

the first (and last) element(s). As a consequence, even if the list is ordered, we cannot use binary search either.

10.3.1 Operations

The linked list supports operations search (linear search only), insert, delete.

10.3.1.1 Search

We check the key of each element and take the next until we arrive at the end of the list or find key k. Aswe only use forward pointers, the given Algorithm 10.8 works in this same form both for singly and doubly linked lists. In worst case, we have to check all elements, hence it has runtime O(n), when n is the number if currently stored elements (which is not stored as an attribute of the list).

Algorithm 10.8 Search in (doubly) linked list

```
1: LL_SEARCH(L,k,@x)
2: // INPUT: L linked list, key k to search
3: // OUTPUT: x pointer to an element with key[x] = k, or NIL if no such element exists
4: x ← head[L]
5: WHILE (x ≠ NIL) AND (key[x] ≠ k) DO      // Again assume second condition is not checked if
   the first fails.
6:     x ← next[x]
7: RETURN(x)      // We have either stopped because x is NIL, that is we hit the end of the list,
   or because we have found key[x]=k.
```

10.3.1.2 Insert

Instead of an index to insert at, we give the pointer of the element to insert *after*. We take note of the element z currently following y, and change the pointers of y, x and z to link the three in this order. Algorithm 10.9 may be shortened, as the some of the same commands are given on the true and false branches; however is given in the more verbose form to remain more readable. For a singly linked list, all prev[] field values may be omitted. As it only deals with neighboring elements, it has runtime

O(1). Another approach is possible, where we always insert at the beginning of the list, which also has runtime O(1). Figure 10.6a shows insertion into a doubly linked list.

Algorithm 10.9 Insert in doubly linked list

```
1:  LL_INSERT(L,x,y)
2:  // INPUT: L linked list, x pointer of new element to insert, y pointer of element to insert after
    (possibly NIL to insert x as first element)
3:  IF y = NIL THEN      // we insert as first element
4:      z ← head[L]      // old first element to insert before
5:      head[L] ← x
6:      next[x] ← z
7:      prev[x] ← NIL
8:      IF x $\neq$ NIL THEN
9:          prev[z] ← x
10: ELSE
11:     z ← next[y]      // x will be inserted between y and z
12:     next[y] ← x
13:     next[x] ← z
14:     prev[x] ← y
15:     IF z $\neq$ NIL THEN
16:         prev[z] ← x
17: RETURN()
```

10.3.1.3 Delete

We delete an element x given by its pointer. As we must link the previous element y to the following element z, the given Algorithm 10.10 only works for a doubly linked list. As we only deal with neighboring elements, this operation has runtime O(1). Figure 10.6b shows deletion from a doubly linked list.

To adapt the deletion algorithm for a singly linked list, two approaches are possible. In the first one, we may modify the algorithm and give the pointer of the previous element y instead of x, so that we have access to it, and we can still access x as well as z following the links, still with runtime O(1). Alternatively, we may give the pointer to x and do a search for y, with condition next[y] = x instead of key[y] = k, but that increases the runtime to O(n).

10.3.2 Extensions

10.3.2.1 Sentinel

We talk about one possible extension, a doubly linked list with sentinel ("guard"). (A sentinel can also be introduced into a list with only forward links, that is left to the reader.) The motivation is that handling the NIL pointers at the beginning and end of the list becomes cumbersome; hence instead we re-create NIL as an element, which becomes the sentinel. The sentinel can be recognized by its invalid key, and we link it between the end and start of the list to turn it into a cyclical structure.

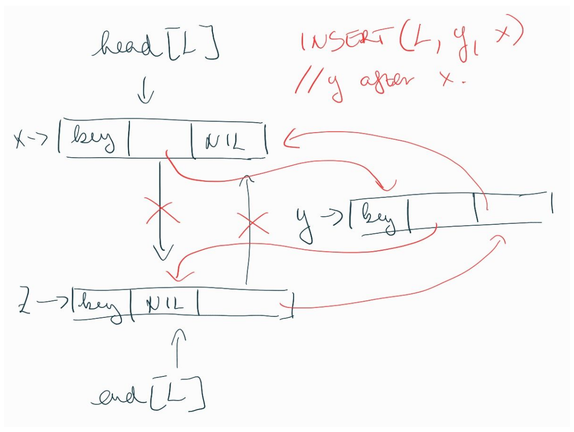
If the first element is a, the last element z, and the sentinel is s, we link them as follows: next[z] = s, prev[a] = s, next[s] = a, prev[s] = z. Instead of the head and end attributes, the list can now be accessed by a single nil[L] = s attribute. If the list is empty, next[s] = prev[s] = s.

Algorithm 10.10 Delete from doubly linked list

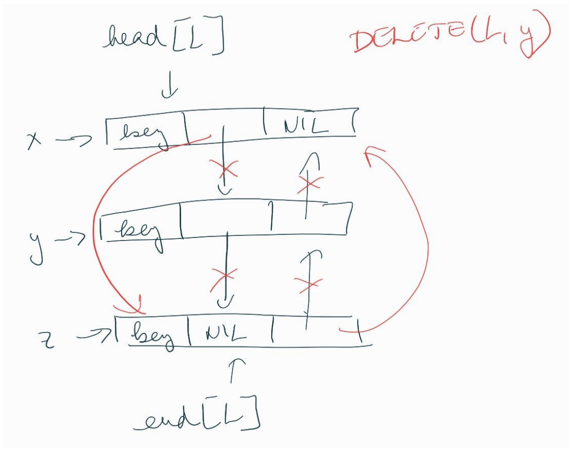
```
1: LL_DEL(L,x)
2: // INPUT: L linked list, x (non-NIL) pointer of an element to delete
3: IF x ≠ NIL THEN
4:     y ← prev[x]
5:     z ← next[x]
6:     IF y = NIL THEN
7:         head[L] ← z
8:     ELSE
9:         next[y] ← z
10:    IF z = NIL THEN
11:        end[L] ← y
12:    ELSE
13:        prev[z]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

Figure 10.6: Insertion into and deletion from a doubly linked list

(a) Insertion



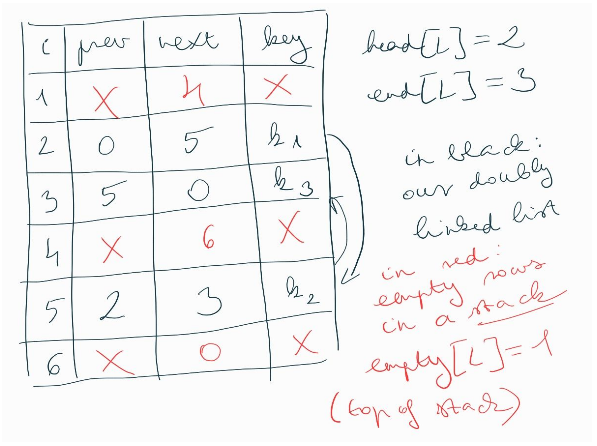
(b) Deletion



10.3.2.2 Array implementation

As a linked list also uses bins with the same structure to store records, we may actually pre-allocate consecutive bins in memory, similar to an array. (However, we lift the restriction that consecutive bins must be used for storage.) The downside is losing the infinite capacity of the linked list, however instead of pointers, it is sufficient to use indices, and allocating memory is handled all at once at the beginning. The head[L] and end[L] attributes are also indices now, and the NIL pointer is replaced by the invalid 0 index. The available, empty bins may be stored in a stack structure (see Section 10.4.2), as a singly linked list again embedded in the array. The top of this stack can be accessed by the empty[L] attribute. Figure 10.7 shows a visual representation of the array implementation of the linked list.

Figure 10.7: Array implementation of linked list



i	prev	next	key
1	X	4	X
2	0	5	k_1
3	5	0	k_3
4	X	6	X
5	2	3	k_2
6	X	0	X

head[L] = 2
end[L] = 3
in black: our doubly linked list
in red: empty rows in a stack
empty[L] = 1
(top of stack)

10.4 Queue and stack

The following queue and stack are special data structures, where insertion and deletion may only happen at predefined points. As such, they are useful for keeping track of data that has to be processed in a certain order during the run of an algorithm.

10.4.1 Queue

The queue is inspired by queuing: the first inserted, "oldest" element must be the first to be removed. As such, the queue has a front and an end, new elements are inserted at the end, slowly trickle to the front as they become older, and old elements leave at the front. We call it a FIFO: *first in first out* structure.

Definition 10.3 (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).

We show two implementations: with array and with linked list.

10.4.1.1 Linked list implementation

We can implement the queue with a singly linked list, with an additional end[L] attribute. The head of the list will serve as the front of the queue where elements can be popped (deleted) from, and the linked second element can step into its place, see Algorithm 10.12. The end of the list serves as the end of the queue, where new elements can be pushed (inserted after the last element), see Algorithm 10.11. The end attribute can be maintained in constant time: it is initialized as NIL together with the head, then whenever a new element is added, end[A] becomes the pointer of this new element. Figure 10.8a shows the linked list implementation of the queue and its operations.

10.4.1.2 Array implementation

The elements of the queue will be stored consecutively within the array, however not (necessarily) starting from the first bin available. This saves us from having to move elements each time one is removed, removing one of the major downsides of the array management. (The downside of finite capacity of course remains, but as a pro, we may again work with indices instead of pointers.)

The head and end attributes will serve to signal where the queue actually begins and ends within the array. More precisely, head is the index of the first element; it is initialized as 1. The end signals

Algorithm 10.11 Insert (push) into 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()
```

Algorithm 10.12 Delete (pop) from queue, with linked list implementation

```
1: QUEUE_POP_LL(Q,@x)
2: // INPUT: Q singly linked list for storing the queue
3: // OUTPUT: x pointer of the first element, or NIL if the queue is empty
4: x ← head[Q]
5: IF x ≠ NIL THEN
6:     head[Q] ← next[x]
7: RETURN(x)
```

the first free index after the queue, where a new element can be inserted, it is initialized as 1. Figure 10.8b shows the array implementation of the queue and its operations.

Let us denote n = arraysize[Q]. We will also use the array cyclically: after index n, index 1 follows again. This will take some management to reset the indices when they overflow. We always want to leave one bin empty between the end and start of the queue, hence our storage capacity is actually $n-1$. We recognize the empty queue by head[Q] = end[Q] and the full queue by end[Q] + 1 = head[Q], or head[Q]=1 and end[Q]=n.

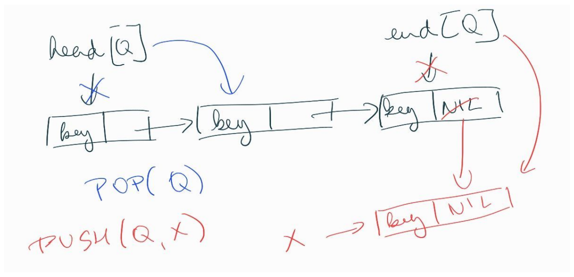
We show the push operation in Algorithm 10.13 and the pop operation in Algorithm 10.14.

Algorithm 10.13 Insert (push) into queue, with array implementation

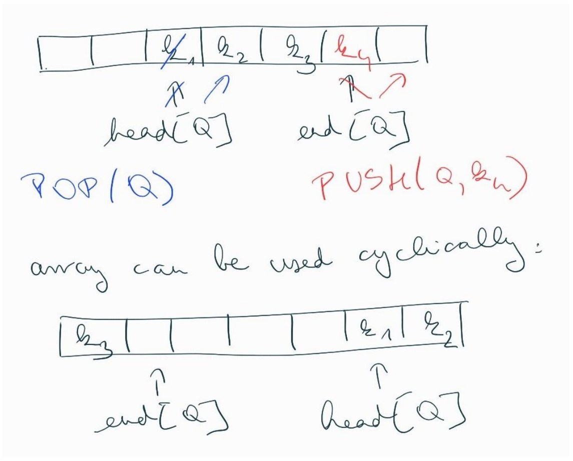
```
1: QUEUE_PUSH_ARR(Q,x,@message)
2: // INPUT: Q array for storing the queue, x new record to insert
3: // OUTPUT: message: success or error message
4: IF head[Q] = end[Q]+1 OR (head[Q] = 1 AND end[Q] = n) THEN     // In short, head[Q] ≡ end[Q]+1  mod  n.
5:     message ← "queue is full"
6:     RETURN(message)
7: $Q_{end[Q]}$ ← x
8: INC(end[Q])
9: IF end[Q] > n THEN     // Check for overflow and cyclically reset
10:     end[Q] ← 1
        // In short for the above 3 lines, we could write end[Q] ← end[Q]+1  mod  n
11: message ← "successfully inserted"
12: RETURN(message)
```

Figure 10.8: Implementations of the queue and its operations

(a) Linked list implementation



(b) Array implementation



Algorithm 10.14 Delete (pop) from queue, with array implementation

```
1:  QUEUE_POP_ARR(Q,@x,@message)
2:  // INPUT: Q array for storing the queue
3:  // OUTPUT: x element, message: error or success message
4:  IF head[Q] = end[Q] THEN
5:      x ← NULL
6:      message ← "empty queue"
7:      RETURN(x,message)
8:  x ← $Q_{head[Q]}$
9:  INC(head[Q])
10: message ← "successfully removed"
11: RETURN(x,message)
```

10.4.2 Stack

The stack can be visualized vertically: it only has a top, where new elements can only be inserted at the top, but also only the top element can be removed. Hence it becomes a LIFO: *last in first out* structure.

Definition 10.4 (Stack (data structure))**.** The stack is a dynamic set where insertion and deletion may only happen at a predefined points. We always remove the latest inserted element. Supported operations are push (insert) and pop (remove).

We have already mentioned two common usages of such a structure:

- the stack that handles recursive calls. Clearly, that must be a LIFO structure: only the last halted/paused program instance can continue the work with the partial results returned.
- in the array implementation of the linked list, we can store indices of available bins in a stack structure. While the order does not matter here and other structures would work as well, we will soon see that the stack can be realized with a singly linked list, hence that will be the easiest structure to maintain.

We show two implementations: with linked list and with array.

10.4.2.1 Linked list implementation

We can implement the stack S with a regular singly linked list. The single entry point is the single attribute top[S], that functions the same as the head attribute. When a new element is pushed (Algorithm 10.15), it is inserted as first element, and always the first element is popped (Algorithm 10.16). Figure 10.9a shows the linked list implementation of the stack.

Algorithm 10.15 Insert (push) into stack, with linked list implementation

```
1: STACK_PUSH_LL(S,x)
2: // INPUT: S singly linked list for storing the stack, x pointer of new element
3: y ← top[S]
4: next[x] ← y
5: top[S] ← x
6: RETURN()
```

Algorithm 10.16 Delete (pop) from stack, with linked list implementation

```
1: STACK_POP_LL(S,@x)
2: // INPUT: S singly linked list for storing the stack
3: // OUTPUT: x pointer of the first element, or NIL if the stack is empty
4: x ← top[S]
5: IF $x \neq$ NIL THEN
6:     top[S] ← next[x]
7: RETURN(x)
```

10.4.2.2 Array implementation

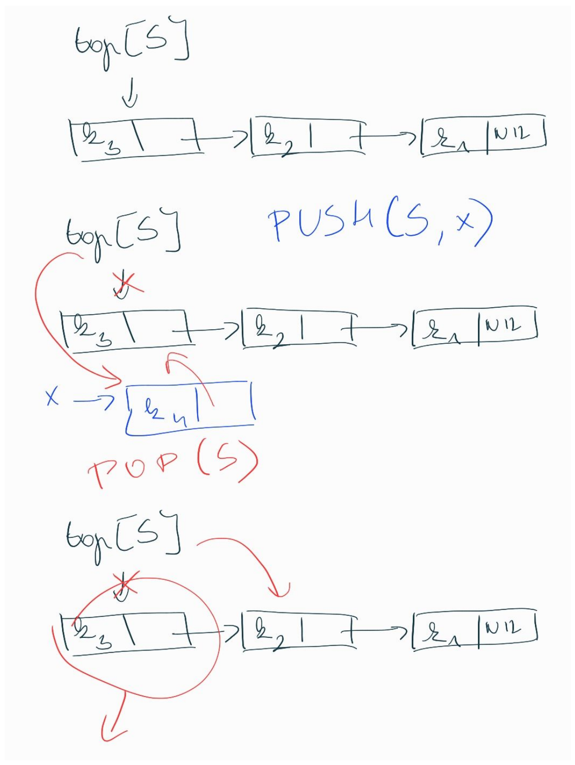
In the array implementation, clearly we lose the infinite capacity, but we do not have to deal with pointers and allocating memory separately for each element. We start filling the bins from index 1 with the oldest elements. We only need the attributes n = arraysize[S], which will be our maximal capacity, and top[S], which is the index of the latest inserted element (serves a similar purpose as the end attribute of the array). We recognize the empty stack by top[S] = 0 and the full stack by top[S] = n. Algorithms 10.17 and 10.18 show the implementations of the push and pop operations. Figure 10.9b shows the linked list implementation of the stack.

Algorithm 10.17 Insert (push) into stack, with array implementation

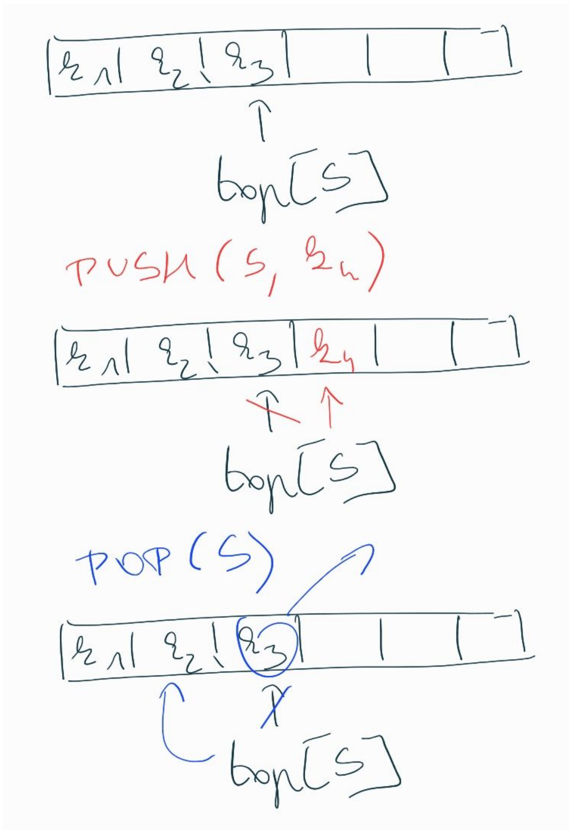
```
1: STACK_PUSH_ARR(S,x,@message)
2: // INPUT: S array for storing the stack, x new element
3: // OUTPUT: message: error or success message
4: IF top[S] = n THEN
5:     message ← "stack is full"
6:     RETURN(message)
7: INC(top[S])
8: $S_{top[S]}$ ← x
9: message ← "successfully inserted"
10: RETURN(message)
```

Figure 10.9: Implementations of the stack and its operations

(a) Linked list implementation



(b) Array implementation



Algorithm 10.18 Delete (pop) from stack, with array implementation

```
 1: STACK_POP_ARR(S,@x,message)
 2: // INPUT: S array for storing the stack
 3: // OUTPUT: x first element, or NIL if the stack is empty, message: error or success message
 4: IF top[S] = 0 THEN
 5:     x ← NIL
 6:     message ← “empty stack”
 7:     RETURN(x,message)
 8: x ← $S_{top[S]}$
 9: DEC(top[S])
10: message ← “successfully removed”
11: RETURN(x,message)
```

Chapter 11

Lecture: hash table

Back to top

Contents of chapter

11.1 Motivation and efficiency

The hash table is a data structure, which is still a dynamic set, but unlike the array and linked list, is not a sequence. That is, we give up the linear order of the elements. Instead, we introduce a different storage logic, where the so-called hash function directly maps the key to an index, and that is the row of the hash table where we would ideally store that record. (We will introduce the conditions, the hash function, the possible problems and solutions soon.) Think of the hash function like a library cataloging software: you give details of the book you're looking for, and it tells you the location you should find it. This will allow us to search, insert and delete records in O(1) time under ideal conditions. Before diving into how we achieve all this, let us show a comparison table of these operation on the array, linked list, and hash table in Table 11.1.

11.1.1 What are hash tables?

They are a data structure, more precisely, a dynamic set. We store data records in an array-like table, however we do not care about the order of the elements relative to each other (not a sequence!), and empty rows are allowed wherever. The placements of records are determined individually: there is a so-called hash function that maps the key field of the record to an index where we should (ideally) store the record. (Of course, we have to have a plan B in case that row is already occupied, so there is still some interaction between records. We will come back to this later.) Under ideal conditions,

Table 11.1: Operations of array, linked list and hash tables

Operation	Array	Linked list	Hash table
storage logic	linear order	linear order	hashing
search	linear search O(n) binary search O(log(n))	only linear search O(n)	O(1)
insertion	O(n)	O(1)	O(1)
deletion	O(n)	O(1)	O(1)

this handling of records (mostly) independently of each other will allow for faster operations. A hash table supports search, insertion and deletion. Modifying a record is possible if the key field does not change; if the key field changes, we must delete and re-insert the whole record according to the new key (otherwise the search will break down).

When can/do we use hash tables?

- each record must have its *unique* key
- we expect to have a small, effectively random selection of keys, from a much larger set of possibilities (feasible set)
- for example, a company has about 100 employees, and wants to identify them by their TAJ number (the Hungarian health care ID), which is a 9 digit number, and thus has 10^9 possible values
- for simplicity, we will assume the keys k come from $1, 2, \ldots, M$
- suppose we expect we will have n records to store, which is *much* smaller than M (thus it is not feasible to have a mostly empty table of size M and store each record on the index that is equal to its key)
- we calculate the size of the hash table with a little wiggle room, usually 20% extra, $N \approx 1.2 \cdot n$. We index the rows of the hash table by $0, 1 \ldots, N-1$.

What is hashing?

- hashing is mapping the key into an index of the hash table. This is done by a deterministic *hash function* $h : \{1, 2, \ldots, M\} \to \{0, 1, \ldots, N-1\}$.
- since we are mapping a much larger set into a smaller one, *inevitably*, there will be identical values
- but that is only a problem, when two different keys present in our dataset have the same hash (same value by the hash function) – we call this a *conflict*
- we do not know in advance which keys will occur (we might as well think of them as random), but we would like to choose the hash function smartly, to have as few conflicts as possible (on average)
- as a rule of thumb, we want to map (roughly) the same amount of keys onto each index
- usually, we also prefer nearby keys to be mapped onto different indices
- the most common hash function that satisfies all the above conditions is $h(k) = (k \mod N)$

How do we deal with conflicts?

- reminder: conflict is when the keys of several records map onto the same index
- the hash value shows the index where we would *ideally* store those records, but clearly, one row can only contain one record, and as such, the latecomer must be placed elsewhere. But we must establish a way to find it in later searches!
- there are in fact different ways to handle conflicts, that lead to different variations or versions of hash tables. We quickly introduce the ideas of all in text, then later we discuss the details (introduce metadata fields, show visual representation, and exact algorithms for the operations).
- **linked list**: we start the search at the index given by the hash function, then continue along a linked list starting in that row. There are two approaches here.
  - *not embedded linked list*: in this version, in fact no records are stored in the hash table, only pointers to N many linked lists. Records with $h(k) = i$ are all stored in a linked list L_i, and the hash table only stores the head$[L_i]$ pointers (possibly NIL). This has the advantage of having basically unbounded capacity, in case we have underestimated the number of records n, however when n becomes much larger than the size of the table N, the linked lists become long and search becomes slow again.
  - *embedded linked list*: in this version, all records are stored in the hash table. Suppose we want to place a record with key k in row $i = h(k)$, but row i is already occupied. We search for an empty row starting from the end of the table, place the record k there, and make it searchable by adding it to the linked list that starts at row i. This linked list is embedded into the hash table, similarly to the array implementation of the linked list. The downside of this approach is that these linked lists may grow into each other and become long, hence search may become slow again. The situation gets worse as the table fills up more and more.
- **open address**: instead of a single hash value and then searching along linked lists, this approach turns the hash function into a two-variable function $h(k, t)$, where k is still the key, but t is now the number of the trial. For any fixed key k, for the various trials $t = 0, 1, \ldots, N-1$, the hash function $h(k, t) = i$ should take all different indices i in $\{0, 1, \ldots, N-1\}$ in some order, which defines a *search sequence*. We can think of it as the "preference order" in which we try to insert the record k into various rows i. We keep the previous one-variable hash function $h_0(k) = k \mod N$ as the starting point of the search sequence given by $h(k, 0)$. There are a couple different approaches to defining the two-variable hash function.
  - *linear trial*: we trial $i = h_0(k)$ first, then start increasing the index i one by one (restarting from the beginning when we reach the end of the table), until we find an empty row. Formally, we use hash function $h(k, t) = h_0(k) + t \mod N$. More generally, we could use $h(k, t) = h_0(k) + c \cdot t \mod N$, increase the index by c in each step, for some integer $1 \leq c < N$. We must also have $\gcd(c, N) = 1$ so that the search sequence includes all rows.
  - *quadratic trial*: we trial $i = h_0(k)$ first, then start increasing the index i first by 1, then the new index by 2, then the new index by 3, and so on. Formally, we use hash function $h(k, t) = h_0(k) + \frac{t(t+1)}{2} \mod N$. The aim of taking larger and larger steps is to hopefully quickly leave *clusters*: when many consecutive rows are occupied.
  - *double hash*: we again trial $i = h_0(k)$ first. But to avoid clustering, we want to create different search sequences for different keys, even if their starting point given by h_0 was the same. We will do linear searches for each, but with different stepsize c, given by a second function $h_1(k) = 1 + (k \mod (N-1))$, which takes values in $1, 2, \ldots, N-1$. That is, the two-variable hash function is $h(k, t) = h_0(k) + t \cdot h_1(k) \mod N$.

Operations

Hash tables of all kinds support operations search, insert and delete.

Quite obviously, to delete a record with a given key, we must search and find the given record. To maintain the **uniqueness of keys**, insertion must also start with a search, to make sure the key is not already in the hash table. If it is already there, we will give an error message, and the user may wish to edit the record instead.

11.1.2 Hash tables with linked lists

Reminder: with this approach, the hash function gives a single index where we would ideally store the record. In case there are conflicts, we follow the linked list starting (or continuing) at this index to find the record, or we must insert the record into this linked list.

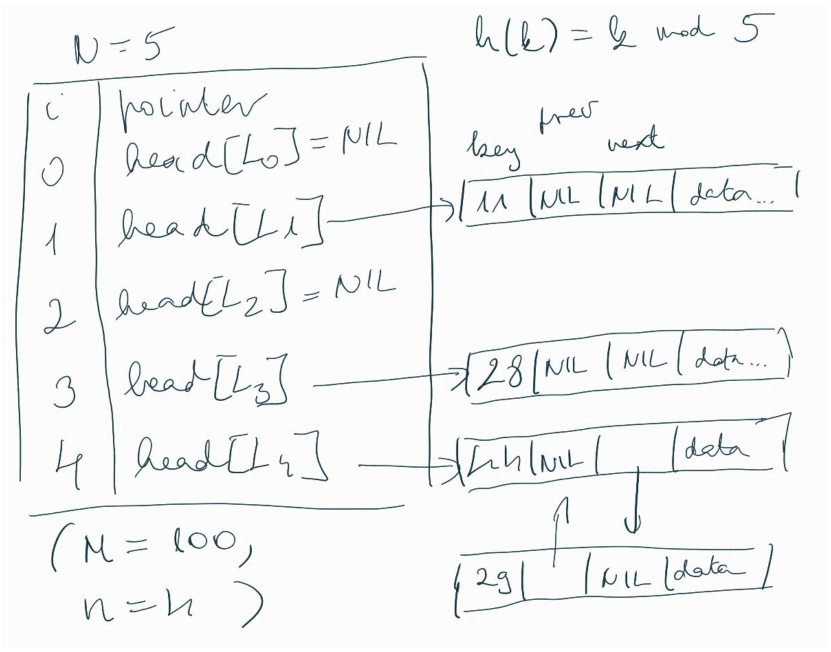
11.1.2.1 Non-embedded variant

Reminder: in this variant, in fact no record is stored in the table itself; the table in row i only stores head$[L_i]$ pointers to separate linked lists. For a record with key k, it must be stored in the list according to $h(k) = i$. Search hence becomes linear search in this linked list, and insertion, after making sure by a search that the key does not exist yet, happens at the beginning of this list. To make deletions fast, we use doubly linked lists.

A table of size N has rows indexed by $i = 0, 1, \ldots, N-1$, and we use hash function $h(k) = k \bmod N$. Figure 11.1 shows a visual representation of such a hash table. In our example, the keys may come from $1, 2, \ldots, M = 100$, and we have to store $n = 4$ of them. We chose table size $N = 5$ for that.

As all hash tables, this variant supports search, insert and delete operations (in fact the latter two also rely on search). We give them in Algorithms 11.1, 11.2 and 11.3, respectively.

Figure 11.1: Hash table with non-embedded linked lists



11.1.2.2 Embedded linked lists

In this approach, all records are stored in the table itself. In case of a conflict, the latecomer record is placed at the end of the table, and we find it along a linked list starting at its preferred index and

Algorithm 11.1 Search in hash table with non-embedded linked lists

```
SEARCH(T,k,@x)
// INPUT: T hash table, k key
// OUTPUT: x pointer of record with key[x]=k or NIL
$i \leftarrow h(k)$
x $\leftarrow$ head[$L_i$] $\leftarrow T_i$     // The pointer $T_i$ is stored in both head[$L_i$] and x.
WHILE (x $\neq$ NIL) AND (key[x] $\neq$ k) DO
    x $\leftarrow$ next[x]
RETURN(x)
```

Algorithm 11.2 Insertion into hash table with non-embedded linked lists

```
INSERT(T,x,@message)
// INPUT: T hash table, x record to insert
// OUTPUT: success or error message
$k \leftarrow$ key[x]
$i \leftarrow h(k)$
y $\leftarrow$ head[$L_i$] $\leftarrow T_i$     // First a search to make sure key $k$ is not in the hash table yet.
WHILE (y $\neq$ NIL) AND (key[y] $\neq$ k) DO
    y $\leftarrow$ next[y]
IF y = NIL THEN     // Key $k$ not found, we can insert.
    LL_INSERT($L_i$,x,NIL)     // See Algorithm 10.9. For simplicity, we insert as first element.
    message $\leftarrow$ "successfuly inserted"
ELSE
    message $\leftarrow$ "key k already exists"
RETURN(message)
```

embedded into the table as well (see array implementation of linked list). As usual, a table of size N has row indices $i = 0, 1, \ldots, N-1$.

Each row stores as metadata the next pointer, initially NIL if the row is not part of any linked list, as well its occupation status: F (free), O (occupied) or D (deleted). We also store a number r that helps us find empty rows at the bottom of the table, its meaning is that all rows with index $i \geq r$ are for sure occupied (however, no guarantees that the rows before r are not).

Figure 11.2 shows a visual representation of such a list. The figure also illustrates how the linked lists may merge, and as such, we only link forward, as the previous element is not always unique.

Its operations search, insert and delete are given in Algorithms 11.4, 11.6 and 11.5, respectively.

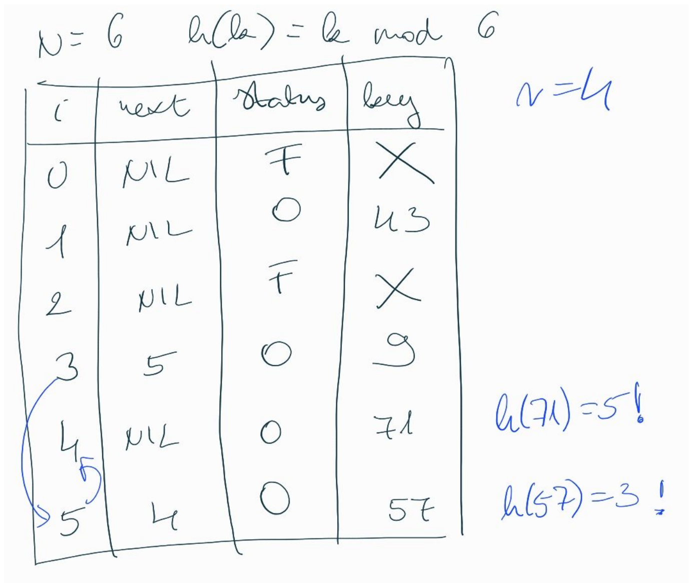
Algorithm 11.3 Deletion from hash table with non-embedded linked lists

```
DELETE(T,k,@message)
// INPUT: T hash table, k key of record to delete
// OUTPUT: success or error message
$i \leftarrow h(k)$
x ← head[$L_i$] ← $T_i$        // First find the record to delete.
WHILE (x ≠ NIL) AND (key[x] ≠ k) DO
    x ← next[x]
IF x = NIL THEN       // Key $k$ not found.
    message ← “key k does not exist”
ELSE
    LL_DEL($L_i$,x)       // See Algorithm 10.10.
    message ← “successfuly deleted”
RETURN(message)
```

Algorithm 11.4 Search in hash table with embedded linked lists

```
SEARCH(T,k,@x)
// INPUT: T hash table, k key
// OUTPUT: i index of record with key[$T_i$]=k or NIL
$i \leftarrow h(k)$
WHILE (i ≠ NIL) AND ((status[$T_i$] = D) OR (status[$T_i$] = O AND key[$T_i$] ≠ k)) DO        // When
we stop, either key[$T_i$] = k, or we reached the end of the linked list.
    i ← next[$T_i$]
RETURN(x)
```

Figure 11.2: Hash table with embedded linked lists



$N = 6 \quad h(k) = k \bmod 6$

$n = 4$

i	next	status	key
0	NIL	F	X
1	NIL	O	43
2	NIL	F	X
3	5	O	9
4	NIL	O	71
5	4	O	57

$h(71) = 5!$

$h(57) = 3!$

For our example, we assume keys come from $1, 2, \ldots, M = 100$, and we store $n = 4$ records in a table of size $N = 6$. We have built the hash table with keys (records) arriving in the order $9, 43, 57, 71$. We have a conflict as $h(57) = 3 = h(9)$, hence 57 is instead placed in row 5. However, afterwards 71 arrives with $h(71) = 5$, which causes a new conflict, and 71 itself must be placed in row 4 instead. This demonstrates how the linked lists can merge. At this snapshot in time, this table has $r = 4$ (only rows 5 and 4 were filled for conflicts, and while row 3 is occupied otherwise, we have not yet checked it while looking for an empty row from the bottom).

Algorithm 11.5 Deletion from hash table with embedded linked lists

```
DELETE(T,k,@message)
// INPUT: T hash table, k key of record to delete
// OUTPUT: success or error message
$i \leftarrow h(k)$
p ← NIL
WHILE (i $\neq$ NIL) AND ((status[$T_i$] = D) OR (status[$T_i$] = O AND key[$T_i$] $\neq$ k)) DO        // Previous search with the same modification.
    p ← i
    i ← next[$T_i$]
IF i = NIL THEN      // Record k not found.
    message ← “key k not found”
ELSE
    status[$T_i$] ← D
    IF p $\neq$ NIL THEN
        next[$T_p$] ← next[$T_i$]      // No other pointers are changed!
    IF r $\leq$ i THEN
        r ← i+1
    message ← “successfully deleted”
RETURN(message)
```

Algorithm 11.6 Insertion into hash table with embedded linked lists

```
INSERT(T,x,@message)
// INPUT: T hash table, x record to insert
// OUTPUT: success or error message
$k \leftarrow$ key[x]
$i \leftarrow h(k)$
IF status[$T_i$] = F THEN      // No linked list yet, easy case.
    $T_i \leftarrow$ x
    status[$T_i$] $\leftarrow$ O
    message $\leftarrow$ "successfully inserted"
    RETURN(message)
d $\leftarrow$ NIL
WHILE (i $\neq$ NIL) AND ((status[$T_i$] = D) OR (status[$T_i$] = O AND key[$T_i$] $\neq$ k)) DO      // The search algorithm, modified.
    IF D = NIL AND status[$T_i$] = D THEN
        d $\leftarrow$ i      // Remember the first deleted row.
    p $\leftarrow$ i      // We always remember the previous element, so that when i=NIL, we remember the last element.
    i $\leftarrow$ next[$T_i$]
IF i $\neq$ NIL THEN      // If k was found
    message $\leftarrow$ "key k already exists"
ELSE      // k is not found, we can insert.
    IF d $\neq$ NIL THEN      // If a deleted row was seen, we use it. No need to change pointers now.
        $T_d \leftarrow x$
        status[$T_d$] $\leftarrow$ O
        message $\leftarrow$ "successfully inserted"
    ELSE      // Otherwise find a non-occupied row at the end of the table.
        REPEAT
            DEC(r)
        UNTIL r < 0 OR status[$T_r$] $\neq$ O
        IF r < 0 THEN
            message $\leftarrow$ "table is full"
        ELSE      // must have found status[$T_r$] $\neq$ O
            $T_r \leftarrow$ x
            next[$T_p$] $\leftarrow$ r      // p is the previous last element of the linked list
            status[$T_r$] $\leftarrow$ O
            message $\leftarrow$ "successfully inserted"
RETURN(message)
```

11.1.3 Hash tables with open address

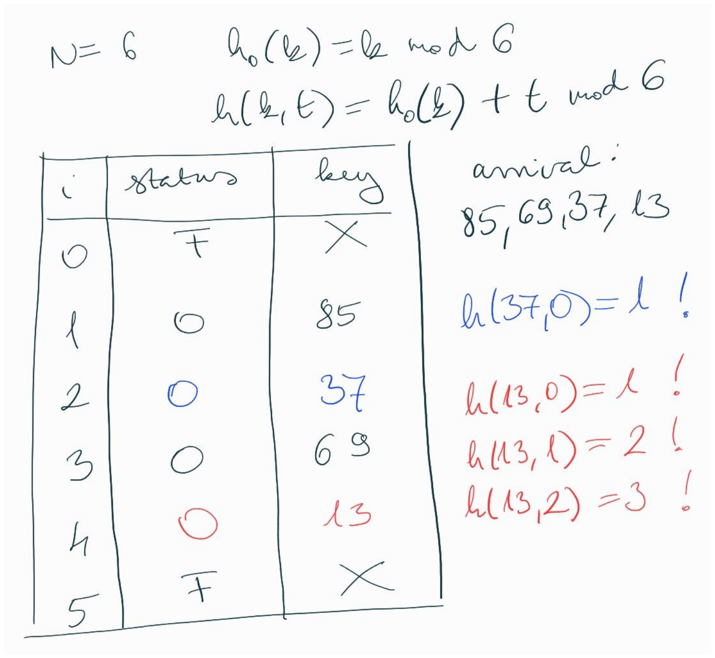
Reminder: open address means the hash function h takes two variables, the key k as well the number of trial $t = 0, 1, \ldots, N-1$. We must define the two-variable h function smartly, so that for any given key k, the so-called search sequence $h(k, 0), h(k, 1), \ldots, h(k, N-1)$ contains all possible indices $i = 0, 1, \ldots, N-1$ in some order (is a permutation of them). We search for the record with key k along this sequence. The previous one-variable hash function serves as our starting point: $h(k, 0) = h_0(k) = k \mod N$. It is important that the search sequence includes each possible index i once: as long as the hash table is not full, we want to find an empty row to insert our record, and we also do not want to waste time checking the same row twice.

The only metadata a hash table with open address stores is the status of each row: F for free, O for occupied, or D for deleted. The deleted status is clearly different from the occupied one: we can place new records in deleted rows. However we must distinguish them from free rows as well, because a record with key k might have been placed later down its search sequence because an earlier row, say $h(k, 0) = i$ was occupied at the time. When we delete the record from this row i, we must then indicate that something has been there before, so we know that the search for k must continue.

The sub-variants of open address hash tables only differ in how the two-variable function $h(k, t)$ is defined and calculated, and we will make use of this to give the algorithms in a common, general form later, after discussing the subtypes and their specific assumptions.

We show an example of an open address hash table in Figure 11.3.

Figure 11.3: An open address hash table



$N = 6$ $\quad h_0(k) = k \mod 6$

$h(k,t) = h_0(k) + t \mod 6$

i	status	key
0	F	X
1	O	85
2	O	37
3	O	69
4	O	13
5	F	X

arrival: 85, 69, 37, 13

$h(37,0) = 1$!

$h(13,0) = 1$!

$h(13,1) = 2$!

$h(13,2) = 3$!

We use table size $N = 6$ and hash function $h(k,t) = (h_0(k) + t) \mod N$. In particular, this is linear trial. We store 4 keys, which arrive in the order 85, 69, 37, and 13. We have indicated all the collisions in color. We see that to place the latecomer 13, we must check the already occupied rows 1, 2 and 3, before we find the empty row 4. This is also an example of clustering and how it can slow down operations in a hash table.

11.1.3.1 Linear trial

Linear trial means a hash function of the form $h(k,t) = h_0(k) + c \cdot t \mod N$. Often $c = 1$, that is, we increase the index one by one and check each row, then loop back to the top of the table when the bottom is reached.

For the search sequence to include all indices once, $\gcd(c, N) = 1$ is necessary. Otherwise, if for example $c = 2$ and N and $h_0(k)$ are both even numbers, we would only search even rows. Clearly, $c = 1$ always works.

The drawback of such a search sequence is clustering: when many consecutive rows are occupied, it takes a long time to find an empty one again.

11.1.3.2 Quadratic trial

Most commonly used quadratic hash function is $h(k,t) = \left(h_0(k) + \frac{t(t+1)}{2}\right) \mod N$. Noting that $\frac{t(t+1)}{2} = 0 + 1 + 2 + \cdots + t$, this results in first increasing the index by 1, then by 2, then by 3, and so on. The more general form would be $h(k,t) = \left(h_0(k) + c_1 \cdot t + c_2 \cdot t^2\right) \mod N$ (the specific example above corresponds to $c_1 = c_2 = \frac{1}{2}$).

For the specific case of $h(k,t) = h_0(k) + \frac{t(t+1)}{2} \mod N$, the search sequence gives each index exactly once, exactly when N is a power of 2 (we do not prove this, but as an exercise, try writing the search sequences starting from 0 for all single-digit table sizes $N = 2, 3, \ldots, 9$, and check that indeed sizes 2, 4, 8 work.) For the general case, we do not study what is a good combination of N, c_1 and c_2.

By increasing the index by larger and larger numbers, this approach helps in getting through many consecutive occupied rows faster. However, keys with the same initial hash still follow the same search sequence, which means in the unlucky case of having several keys with the same hash, we still need to follow the search sequence longer and longer, which is called secondary clustering.

11.1.3.3 Double hash

This approach intends to eliminate clustering overall, by making sure different keys even with the same hash will have different search sequences.

This can be achieved quite simply: we do linear search with different stepsize c, which is given by a second function $h_1(k) = 1 + (k \mod (N-1))$. Using a different modulus makes it unlikely that two keys have not only the same h_0, but also the same h_1 values. We add 1 to make sure $c > 1$.

Recall that linear trial requires $\gcd(c, N) = 1$ for the search sequence to give all indices once. Our stepsize $c = h_1(k)$ can and will take all values from 1 to $N - 1$, thus N must be a prime number to be a relative prime to all.

11.1.3.4 Operations

As before, open address hash tables have operations search, insert and delete (the latter two also relying on search). As the only difference between subtypes is in the specific calculation of $h(k,t)$, we write their algorithms together in a general form. We signal the specific steps for each subtype with colors:

- general calculation to be replaced – gray
- linear trial – red
- quadratic trial – blue
- double hash – green

The operations are of course defined in a matching way, maintaining the same agreed upon conventions.

The search. The main idea is that we would like to place each element as early along its search sequence as possible. When we search for a key k,

- if we see an occupied row with key k, clearly we have found it.
- if we see an occupied row with a different key, the search must continue – maybe this key was already here earlier than k and forced us to place k later along its search sequence.
- if we see a deleted row, we also must continue the search – maybe something *was* here earlier, and forced us to place k later along its search sequence.

- if we see a free row, the search can stop and we conclude k is not in the table – because if it was, it should have been placed as early along the search sequence as possible, so earlier, or at the latest here in this empty row.
- to be entirely precise, we also check if $t = N$ already – which means we have checked all rows, and have not found k. (This can happen when there's no more free rows, but many deleted ones – see Section 11.1.3.5 on how to fix this.)

The search is given in Algorithm 11.7.

Algorithm 11.7 Search in open address hash table

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

Insert. Insertion is based on the search. To avoid duplicates, a search must first be completed to make sure that k is not already in the table. However, instead of calling the search algorithm, we use a modified version. Here, it is useful to take note of deleted rows during the search, as we would like to place the record as early along the search path as possible, and both deleted and free rows can be used for that. The insertion is given in Algorithm 11.8.

Delete. To delete an element, we must first find it. Once found, we simply set the row's status to deleted. We will now consider the stored key and data as memory junk, which will be replaced by new data when the row is re-used. The deletion is given in Algorithm 11.9.

11.1.3.5 Discussion

As mentioned at the search, we have to be prepared for when all rows are deleted or occupied. This can happen once we have deleted some records and inserted some more. As we have defined the operations, all rows start as free, but once they have been occupied, they only go back and forth between occupied and deleted status. This can become a problem, since an unsuccessful search can only stop when it encounters a free row, or once it has checked all rows. The latter of course takes a linear runtime, which is against the purpose of the hash table.

Hence, once the number of free rows becomes very small, it is worth it to start anew with an empty hash table (we might even consider a larger size), and "copy" over all current records (insert them one by one, taking care of conflicts). This is a one-time linear cost, but once again reduces the runtime of any and all further operations from linear to constant.

Algorithm 11.8 Insert into open address hash table

```
INSERT(T,x,@message)
// INPUT: T hash table, x record
// OUTPUT: success or error message
k ← key[x]
$t \leftarrow 0$
i ← $h_0(k)$
d ← NIL
s ← $1 + (k \mod (N-1))$      // $h_1$ function, stepsize – for double hash
WHILE $t \leq N - 1$ AND (status[$T_i$] = D OR (status[$T_i$] = O AND key[$T_i$] ≠ k)) DO
    IF d = NIL AND status[$T_i$] = D THEN
        d ← i      // Remember the first deleted row.
    INC($t$)
    $i \leftarrow h(k,t)$      // general form, to be replaced
    i ← $i + c$      // linear trial, with general stepsize $c$
    i ← $i + t$      // quadratic trial, specific case
    i ← $i + s$      // double hash
IF status[$T_i$] = O AND key[$T_i$] = k THEN
    message ← "key k already exists"
ELSE      // We can insert, if there is space.
    IF d ≠ NIL THEN      // If deleted row was seen, we use it.
        $T_d$ ← x
        status[$T_d$] ← O
        message ← "successfully inserted"
    ELSE IF t = N THEN
        message ← "table is full"
    ELSE      // i must be the first free row along the search sequence
        $T_i$ ← x
        status[$T_i$] ← O
        message ← "successfully inserted"
RETURN(message)
```

Algorithm 11.9 Delete from open address hash table

```
DELETE(T,k,@message)
// INPUT: T hash table, k key
// OUTPUT: success or error message
i ← SEARCH(T,k)
IF i = NIL THEN
    message ← "key k does not exist"
ELSE
    status[$T_i$] ← D
    message ← "successfully deleted"
RETURN(message)
```

Chapter 12

Exercises

Back to top

Contents of chapter

12.1 Hash tables

Exercise 12.1 (Open address, linear trial)**.** Consider an open address hash table of size $N = 9$, with hash function given by:

$$h_0(k) = k \bmod N, \qquad h(k,t) = \big(h_0(k) + t\big) \bmod N.$$

Draw the empty table, then insert (or delete, when instructed) the following keys, in order:

66, 59, 37, 49, 22, delete 49, delete 22, 30

Exercise 12.2 (Open address, quadratic trial)**.** Consider an open address hash table of size $N = 8$, with hash function given by:

$$h_0(k) = k \bmod N, \qquad h(k,t) = \Big(h_0(k) + \frac{t(t+1)}{2}\Big) \bmod N.$$

Draw the empty table, then insert (or delete, when instructed) the following keys, in order:

65, 29, 44, 11, delete 44, 72, delete 11, 53

Exercise 12.3 (Open address, double hash)**.** Consider an open address hash table of size $N = 7$, with hash function given by:

$$h_0(k) = k \bmod N, \qquad h_1(k) = 1 + \big(k \bmod (N-1)\big),$$
$$h(k,t) = \big(h_0(k) + t \cdot h_1(k)\big) \bmod N.$$

Draw the empty table, then insert (or delete, when so instructed) the following keys, in order:

67, 9, 28, 37, 23, delete 9, delete 23, 64

12.1.1 A worked out exercise

Exercise 12.4 (Hash table, double hash)**.** Consider an open address hash table of size $N = 7$ and two-variable hash function

$$h(k,t) = \big(h_0(k) + t \cdot h_1(k)\big) \mod N,$$

where

$$h_0(k) = k \mod N, \qquad h_1(k) = 1 + \big(k \mod (N-1)\big).$$

Draw the empty hash table, then execute the following operations: 1. insert 23 (2), 2. insert 26 (5), 3. insert 32 (4), 4. insert 44 (2,2+1 – 1), 5. delete 22 (1; error), 6. insert 30 (2,0+1 – 3), 7. delete 23 (2), 8. insert 30 (error), 9. insert 16 (2,4+1 – 2). Also write down your calculations and the searches you execute!

What to pay attention to:

- all operations rely on a search first!
- **search** continues along the search sequence $h(k,t)$, $t = 0, 1, \dots$, until either:
  1. the key is found,
  2. a free row is found, which means the key is not in the table,
  3. we have checked all rows (should not happen in these exercises)
- to **delete**, must follow the search sequence until the key is found, then delete it – or if not found, error
- all keys must be unique – to **insert**, must first complete the search and make sure the key is not in the table!
  - if key is found, error! no duplicates allowed!
  - after making sure the key is not present, we try to insert it *as early* along the search sequence *as possible*: if a deleted row was seen, we use that; otherwise use the free row where the search stopped
- we show double hash here, because that one causes some confusion sometimes. The logical steps of the solution are the same for linear or quadratic trial as well, except the calculation of $h(k,t)$ becomes simpler.

Solution. For better understanding, we write with more explanation here, this can be noted down in a much more compact way in the midterm. We highlight in blue what we expect to see for full marks in text form in the midterm. For the table, it doesn't need to be drawn again and again, we expect to see the final state in Figure 12.3b, with old states and deleted keys crossed out (but still legible!) to suggest the evolution. Here, we also demonstrate errors (trying to delete a non-existent key and insert a duplicate key), but such cases are not likely to occur in the midterm.

0. Draw the empty hash table, see Figure 12.1a.
1. Insert 23. First calculate partial result:

$$h_0(23) = 23 \mod 7 = 23 - \left\lfloor \frac{23}{7} \right\rfloor \cdot 7 = 23 - 3 \cdot 7 = 2.$$

(Recall the $a \mod b$ operation and the $\lfloor x \rfloor$ function from Definitions 2.6 and 2.1. We won't write this calculation in detail later.) We start the search at

$$h(23,0) = \big(h_0(23) + 0 \cdot h_1(23)\big) \mod 7 = (h_0(23) + 0) \mod 7 = 2.$$

Row 2 is free – search concludes, 23 is not in the table. No deleted row have been seen – use the free row 2 to insert. See Figure 12.1b.

Side note: Note that $h(k,0) = h_0(k)$ always, since we add 0 times something, and $0 \leq h_0(k) < 7$ already, hence $\bmod 7$ just gives back $h_0(k)$ itself. We will quickly use this step without explanation later.

Noting this down quickly:

1. Insert 23.

   $t = 0: \; h(23,0) = h_0(23) = 23 \mod 7 = 2. \Rightarrow$ F.

   23 not found – insert in row 2.

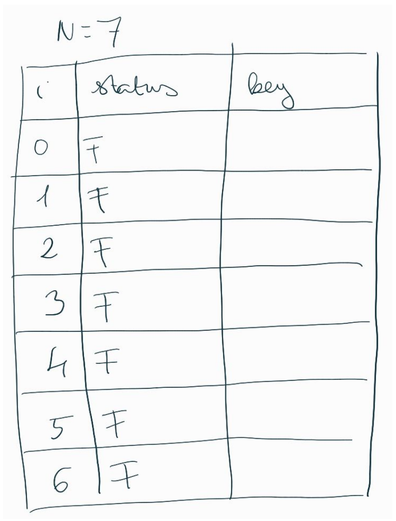
2. Insert 26.

   $t = 0$: $h(26,0) = h_0(26) = 26 \mod 7 = 5 \Rightarrow$ F.

   26 not found – insert in row 5. See Figure 12.1c.

Figure 12.1: Solution of Exercise 12.4, steps 0 to 2.

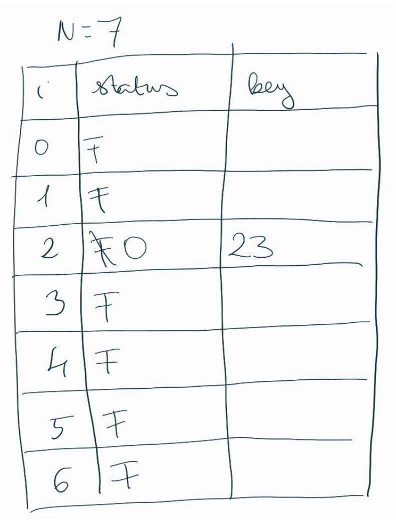
(a) The empty table



N=7

i	status	key
0	F	
1	F	
2	F	
3	F	
4	F	
5	F	
6	F	

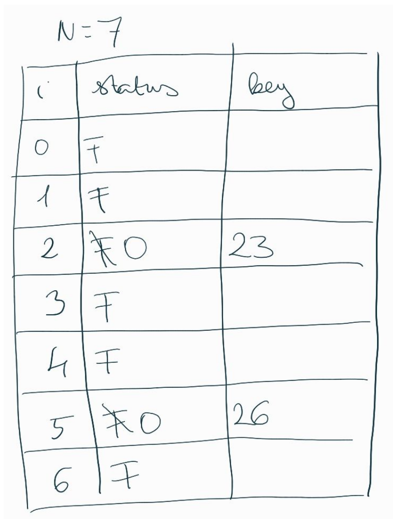
(b) Insert 23



N=7

i	status	key
0	F	
1	F	
2	~~F~~ O	23
3	F	
4	F	
5	F	
6	F	

(c) Insert 26



N=7

i	status	key
0	F	
1	F	
2	~~F~~ O	23
3	F	
4	F	
5	~~F~~ O	26
6	F	

3. Insert 32.

   $t = 0$: $h(32,0) = h_0(32) = 32 \mod 7 = 4 \Rightarrow$ F.

   32 not found – insert in row 4. See Figure 12.2a.

4. insert 44.

   $t = 0$: $h(44,0) = h_0(44) = 44 \mod 7 = 2 \Rightarrow$ O. Row is occupied, check the stored key. $k = 23 \neq 44$ the search must continue.

   To calculate $h(k,t)$ for $t \geq 1$, we're going to need the $h_1(k)$ function as well. The h_1 function shows how many rows we need to jump before the next try – we do not use it directly as an index!

   $h_1(44) = 1 + \big(44 \mod 6\big) = 1 + 2 = 3.$

   $t = 1$: $h(44,1) = (h_0(44) + 1 \cdot h_1(44)) \mod 7 = (2 + 1 \cdot 3) \mod 7 = 5 \mod 7 = 5 \Rightarrow$ O. $k = 26 \neq 44.$ the search still continues!

$t = 2$: $h(44, 2) = (2 + 2 \cdot 3) \mod 7 = 8 \mod 7 = 1 \Rightarrow$ F. search ends here.

44 not found. No deleted rows have been seen, use the free one.

– insert in row 1. See Figure 12.2b.

5. delete 22.

   $t = 0$: $h(22, 0) = h_0(22) = 22 \mod 7 = 1 \Rightarrow$ O. $k = 44 \neq 22$.

   $h_1(22) = 1 + \big(22 \mod 6\big) = 1 + 4 = 5$.

   $t = 1$: $h(22, 1) = (1 + 1 \cdot 5) \mod 7 = 6 \Rightarrow$ F.

   22 not found. We cannot delete if it's not in the table. error! Table does not change (no new figure).

6. insert 30

   $t = 0$: $h(30, 0) = h_0(30) = 30 \mod 7 = 2 \Rightarrow$ O. $k = 23 \neq 30$

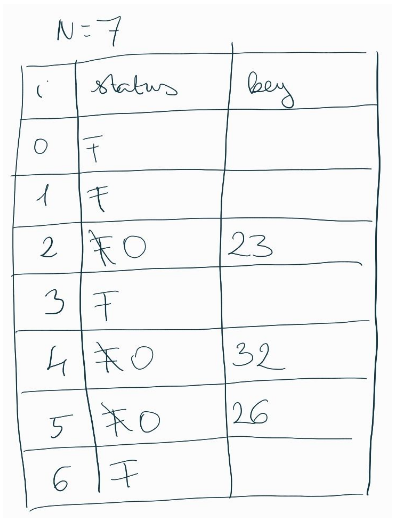
   $h_1(30) = 1 + (30 \mod 6) = 1 + 0 = 1$.

   $t = 1$: $h(30, 1) = \big(h_0(30) + 1 \cdot h_1(30)\big) \mod 7 = (2 + 1 \cdot 1) \mod 7 = 3 \mod 7 = 3 \Rightarrow$ F.

   30 not found, insert into row 3. See Figure 12.2c.

Figure 12.2: Solution of Exercise 12.4, steps 3 to 6.

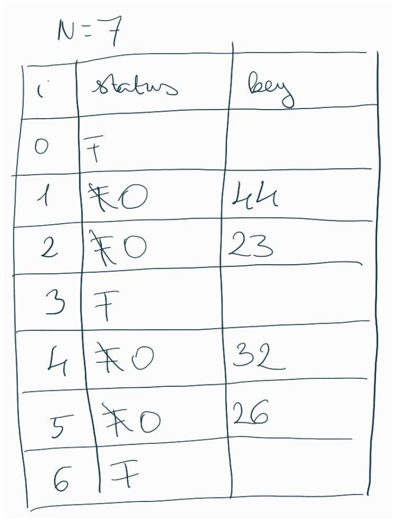
(a) Insert 32



N=7

i	status	key
0	F	
1	F	
2	~~F~~ O	23
3	F	
4	~~F~~ O	32
5	~~F~~ O	26
6	F	

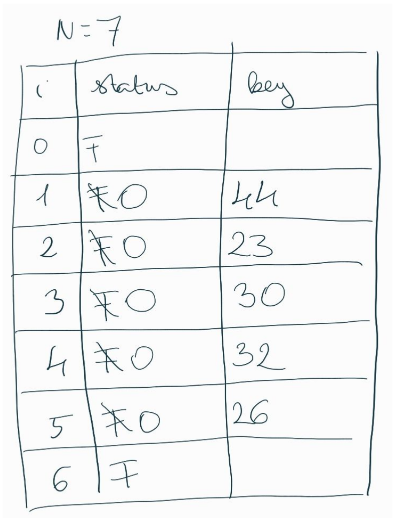
(b) Insert 44



N=7

i	status	key
0	F	
1	~~F~~ O	44
2	~~F~~ O	23
3	F	
4	~~F~~ O	32
5	~~F~~ O	26
6	F	

(c) Insert 30



N=7

i	status	key
0	F	
1	~~F~~ O	44
2	~~F~~ O	23
3	~~F~~ O	30
4	~~F~~ O	32
5	~~F~~ O	26
6	F	

7. delete 23.

   $t = 0$: $h(23, 0) = h_0(23) = 23 \mod 7 = 2 \Rightarrow$ O. $k = 23$

   23 is found, we delete. See Figure 12.3a.

8. insert 30.

   $t = 0$: $h(30, 0) = h_0(30) = 30 \mod 7 = 2 \Rightarrow$ D. Search must continue when we see a deleted row. It's possible the key is later along its search sequence, since something might have been here when we placed it! As is the case now, actually.

   $h_1(30) = 1 + (30 \mod 6) = 1 + 0 = 1$.

   $t = 1$: $h(30, 1) = \big(h_0(30) + 1 \cdot h_1(30)\big) \mod 7 = (2 + 1 \cdot 1) \mod 7 = 3 \mod 7 = 3 \Rightarrow$ O $k = 30$.

30 is found. We cannot have duplicate keys, we cannot insert again! error! Table does not change (no new figure).

9. insert 16.

   $t = 0$: $h(16, 0) = h_0(16) = 16 \mod 7 = 2 \Rightarrow$ D.

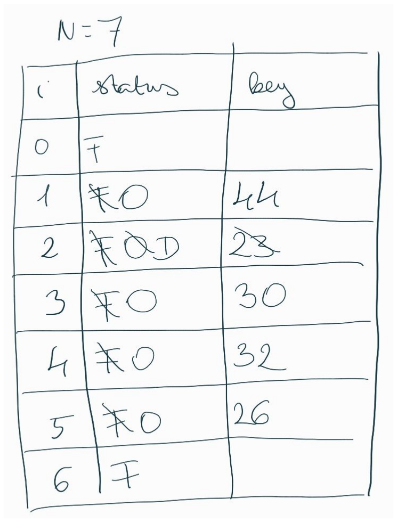
   $h_1(16) = 1 + (16 \mod 6) = 1 + 4 = 5$

   $t = 1$: $h(16, 1) = \big(h_0(16) + 1 \cdot h_1(16)\big) \mod 7 = (2 + 1 \cdot 5) \mod 7 = 7 \mod 7 = 0 \Rightarrow$ F.

   16 not found. We want to insert as early along the search path as possible, and we *have* seen a deleted row. We use that. insert into row 2. See Figure 12.3b.

Figure 12.3: Solution of Exercise 12.4, steps 7 to 9.

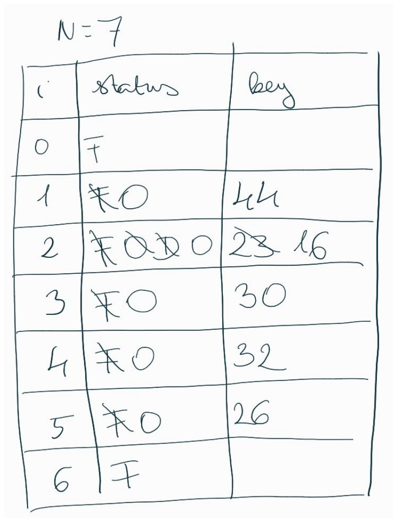
(a) Delete 23



N=7

i	status	key
0	F	
1	~~F~~ O	44
2	~~F~~ ~~O~~ D	~~23~~
3	~~F~~ O	30
4	~~F~~ O	32
5	~~F~~ O	26
6	F	

(b) Insert 16



N=7

i	status	key
0	F	
1	~~F~~ O	44
2	~~F~~ ~~O~~ ~~D~~ O	~~23~~ 16
3	~~F~~ O	30
4	~~F~~ O	32
5	~~F~~ O	26
6	F	

Part V

Miscellaneous

Bibliography

[1] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*. MIT press, 2022.

[2] Attila Házy and Ferenc Nagy. *Adatstruktúrák és algoritmusok*. https://www.uni-miskolc.hu/~matha/adat_alg_NF_HA.pdf.