Fundamentals of Informatics: Lecture Notes

Department of Mathematics & Computer Science
Eindhoven University of Technology (TU/e)

Preface

These lecture notes present the relevant concepts, relationships between these concepts, terminology, and notations (also known as an ontology) for the course Fundamentals of Informatics (2IS80). It is not intended as self-contained study material. In particular, hardly any examples and exercises have been included (for that see the practice assignments).

Material labeled **Questions**, appearing in some sections, shows what kinds of questions you are expected to be able to answer.

Material labeled **Extra**, appearing at the end of most sections and occasionally elsewhere, is not required for the final test. But this extra material can help improve understanding of the required material.

These lecture notes replace the reader [3, various chapters] and the book [2] that were used in previous years. Chapters 3 and 5 currently have minimal content, because these topics were dropped as of 2018–2019 (they are covered more extensively in [2]).

Most references to external material are clickable when reading this document with a PDF viewer. See for instance the link below to the Wikipedia article 'Ontology (information science)'.

N.B. Information available on Wikipedia must be read with a critical mindset. Links to Wikipedia articles are given only for convenience, and not as endorsement of their content.

Feedback is welcome, and can be mailed to `T.Verhoeff@tue.nl`.

Wikipedia Ontology (information science)

Contents

1 Introduction

Informatics studies information and its storage, communication, and processing, in the abstract, that is, without worrying about physical carriers. Information processing is also known as computation, which, despite its name, does not deal exclusively with numbers.

Although computers are relatively young in the history of technology, the concepts of information and computation have played an important role in human history, dating back to when languages first evolved. See Sections 1, 2, 5, and 6 of [8]. And well before that time, biological systems already incorporated features that we now associate with information and computation [1]. Nowadays, even physics has theories that describe the universe as one huge computation operating on information as the fundamental stuff of nature.

An important breakthrough in the automation of communication, storage, and processing of information has been its almost universal **digitization**, driven by our ability

- to convert between (continuous) analog signals and (discrete) digital signals;
- to connect the computational and the physical world through various **sensors** and **actuators**;
- to construct reliable and efficient digital system.

Extra Terminology variants: informatics, computer science, computing science, datalogy.

Wikipedia Computer science — Digitization — Analog-to-digital converter — Digital-to-analog converter — Sensor — Actuator

1.1 Overview

The lecture notes follow the division of the course into four themes:

Models of computation concerns the fundamental nature of computation: what does it take to 'compute' and how do various computational mechanisms differ in computational power;

Algorithms (dropped) concerns the design of effective and efficient solutions to computational problems;

Information concerns the fundamental 'stuff' that computations operate on, focusing on (limits to) efficient, reliable, and secure communication and storage of information;

Limits of computation (dropped) concerns fundamental limits to what can be computed, in principle and in practice.

The material for SAT/SMT solving and program correctness (these topics were added in 2018–2019 to replace algorithms and computability) are covered elsewhere.

2 Models of Computation, Computational Mechanisms

Definition A **model of computation** is a systematic way of (a) defining a class of **computational mechanisms** and (b) reasoning about their execution. Over the years, many models of computation have been introduced and extensively studied. In this course, we deal with only a few of them: finite automaton (§2.4), Turing machine (§2.9), and random-access machine (§2.12).

A model of computation specifies a collection of computational mechanisms, where each mechanism involves an interface between the computation and its environment for input and output (I/O), and computational steps (actions, operations) possibly with an associated cost (such as execution time). The definition of a specific computation under that model is a *syntactic entity*. The syntax tells you what such a definition looks like, and how to write it down. That may involve a special notation (textual or graphical). The *semantics* (meaning) of the defined computation concerns the execution (behavior, dynamics) of that computation. This involves both internal steps and the interaction with the environment.

Note that there are typically three levels involved in the definition of a model of computation:

Model of computation defines the collection of all possible computational mechanisms (such as the collection of all finite automata)

Computational mechanism defines a specific instance of a computational model (a specific finite automaton)

Execution defines how a specific computational mechanism within the model operates in a specific context (such as the execution of a specific finite automaton for a specific input sequence).

Note that each model of computation admits multiple computational mechanisms, and each computational mechanism admits multiple executions, depending on internal decisions (think of a random number generator) and on the behavior of its environment.

Purposes The study of computational mechanisms serves several purposes:

- Understand the nature of computation.
- Help select and build physical devices that compute.
- Help in designing efficient computations.
- Understand the limits of computability.

Extra Terminology variants: model of computation, computational mechanism, computational formalism, computing device

Wikipedia Model of Computation — Computation — Theory of Computation

2.1 Computational Problem

Definition A **computational problem**, in a restricted sense, is any question where the givens (**input**) and possible answers (**output**) are well-defined (mathematical) objects, such as a sequence of symbols (also see §4 on information). The output being well defined also means that the intended relationship between the output and input is well defined.

In a broader sense, a computational problem could involve a dialog, where input and output are interleaved, or even take place in parallel. Such computational problems, however, play a limited role in this course.

A computational problem (in the restricted sense[1]) can be specified by a mathematical function (mapping) or relation from input domain to output domain. Note that such a specification is not the same as a solution to the problem. The problem of sorting a list of words can be specified by these two requirements:

- Each word occurs the same number of times in the output list as that word occurs in the input list. That is, the only possible difference between the input list and the output list is the order of the words.
- The words in the output list appear in ascending lexicographic order.

In a computational problem, the inputs serve as (input) **parameters**. A **problem instance** concerns one specific choice of values for the input parameters. A problem can be considered to consist of the set of all its problem instances.

Note that a computational problem is abstract. Any connection to the real world, involving sensors and actuators, is outside the scope of this concept. Recognizing the number on the license plate of a car and opening the gate if that number is currently authorized is not only a computational problem. The computational problem consists of recognizing a license number from a two-dimensional array of pixel values, and deciding whether that number appears in a data base of authorized numbers. How to obtain the array of pixel values, how to obtain the list of authorized numbers, and how to open the gate is not part of the computational problem.

Wikipedia Computational problem

2.2 Decision Problem

Definition A **decision problem** is a computational problem (§2.1) with a boolean (true/false) output domain, that is, whose answers are yes/no.

Properties

1. The problem of computing $f(x)$ for given x is effectively equivalent with the decision problem $y = f(x)$ where x and y are given, that is, with computing the boolean result $y = f(x)$ for given x and y.
2. A decision problem is completely characterized by the set of inputs for which the output is yes.

[1] Unless stated otherwise, we consider computational problems in the restricted sense.

Wikipedia Decision problem

2.3 (Formal) Language

Definition A **formal language**, or **language** for short, is a set of **symbol** sequences, where symbols are taken from some **alphabet**. A symbol sequence is also known as a **word**. The **empty word**, consisting of no (zero) symbols, is denoted by ε.

Properties Decision problem f on words corresponds to the language $\{ w \mid f(w) \}$, which is the set consisting of all words for which f yields yes. The language V specifies the decision problem (§2.2) 'determine whether $v \in V$ holds, for given input word v'.

Operations Languages can be modified and combined in various ways, such as via the usual set theoretic operations: complement, union, intersection, and difference. Operations specific to languages are concatenation and Kleene closure. The **concatenation** of languages V and W, denoted by VW, is defined as the language consisting of words that are obtained by concatenating any word from V with any word from W (in that order):

$$VW = \{ vw \mid v \in W \wedge w \in W \} \tag{2.31}$$

Note that in VV, the first and second word need not be the same. Also not that in general $VW \neq WV$.
The **Kleene closure** of language V, denoted by V^*, is defined as the language consisting of arbitrary (zero or more, finite) concatenations of words from V:

$$V^* = \{ \varepsilon \} \cup V \cup VV \cup VVV \cup \ldots \tag{2.32}$$

Note that V is contained in V^*; thus, V^* is an extension of V. In fact, V^* is the smallest set X such that

$$\begin{array}{rcll} \varepsilon & \in & X & \text{(zero words from } V\text{)} \\ V & \subseteq & X & \text{(one word from } V\text{)} \\ XX & = & X & \text{(closed under concatenation)} \end{array}$$

Extra In informatics, formal languages are also used to define formal objects other than decision problems. For instance, a programming language is used to define programs, and the HyperText Markup Language (HTML) is used to define web pages. The syntactic structure of such languages is defined by a **formal grammar**. This use of languages is beyond the scope of the course.

Wikipedia Formal language

2.4 Finite Automaton

Definition A **finite automaton** (FA) is a simple computational mechanism with limited computational power (§2.6). Input consists of a sequence of **symbols** taken from a finite input **alphabet**. Output consists of a sequence of yes/no (boolean true/false) values.

During execution of a FA, input symbols are processed one by one; output is produced initially and after receiving each input symbol. This is done as follows. At any moment, the FA is in one of a finite set of **states**, starting in the given **initial state**. One cannot directly observe the current state of a FA; it is a **black box**. The FA's execution is driven by a **transition function**, that determines for each combination of current state and current input symbol what the next state will be. The combination of a state, an input symbol, and a corresponding next state is known as a **transition**. Some states are marked as **accepting state**. The output indicates whether the current state is an accepting state.

Formally, a FA is described by the following five items.

1. A finite set, whose elements are referred to as *states*, serving as identifiers for the FA's states
2. One state marked to be the *initial state*
3. Some states marked to be *accepting states*
4. A finite alphabet, whose symbols are referred to as *input symbols*
5. A function, referred to as *transition function*, that, for every possible combination of current state and input symbol, determines the next state for that combination

The transition function can be given in a *tabular format*, but also graphically in a *diagram*. Such a diagram is a graph with the states as nodes, shown as circles, and the transitions as directed edges (arrows) from current state to next state, labeled by the transition's input symbol. The initial state is marked by an incoming arrow (coming from nowhere), and accepting states are marked by an extra surrounding circle. The state circles can contain the state's identifier, but often these state identifiers are left out, because the circles unambiguously identify the states.

Consider the FA defined by the table in Figure 2.41. It has five states (two of which are accepting

state	initial	accepting	a	b	remark
$q0$	Yes	Yes	$q1$	$q2$	even length, no b
$q1$	No	No	$q0$	$q3$	odd length, no b
$q2$	No	No	$q3$	$q4$	even length, one b
$q3$	No	Yes	$q2$	$q4$	odd length, one b
$q4$	No	No	$q4$	$q4$	more than one b (sink)

Figure 2.41: A table defining a 5-state FA that recognizes even-length words with at most one b

states), input alphabet $\{ a, b \}$, and it recognizes even-length words with at most one b.

A state from which no accepting state can be reached is called a **sink state**. State $q4$ in Figure 2.41 is a sink state.

The diagram in Figure 2.42 defines the same automaton as the table in Figure 2.41.

Two 'parallel' transitions (from the same current state to the same next state) are often merged into one arrow with multiple input symbols as label. The transitions on the sink state (far right) in Figure 2.42 illustrate this.

Finite automata in this course always (must) satisfy the following two constraints[2].

- There exists *exactly one* initial state.

[2]These are so-called *deterministic* FAs, as opposed to *non-deterministic* FAs that have some behavioral freedom.

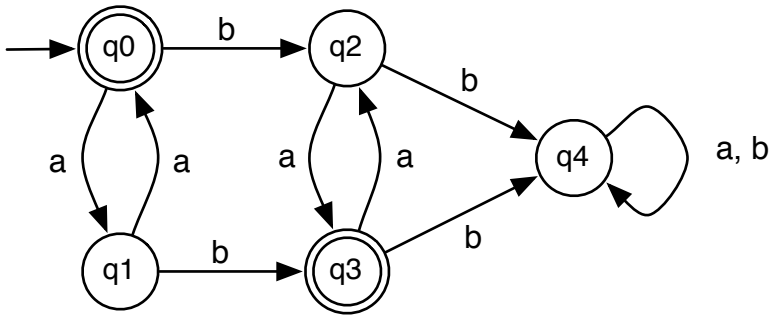

Figure 2.42: A diagram defining a 5-state FA that recognizes even-length words with at most one b

- For each combination of current state and input symbol, there exists *exactly one* transition[3].

Formally, an **execution** or **computation** of a FA for a given input consists of the sequence of states that the FA traverses when processing that input. Figure 2.43 shows the executions of the FA defined

input	state	output
	$q0$	$+$
a	$q1$	$-$
ab	$q3$	$+$
aba	$q2$	$-$
$abaa$	$q3$	$+$
$abaab$	$q4$	$-$
$abaabb$	$q4$	$-$

input	state	output
	$q0$	$+$
b	$q2$	$-$
ba	$q3$	$+$
baa	$q2$	$-$
$baaa$	$q3$	$+$
$baaab$	$q4$	$-$
$baaaba$	$q4$	$-$

Figure 2.43: Two executions for the FA of Figure 2.42 with inputs $abaabb$ and $baaaba$

in Figure 2.42 for the two input words $abaabb$ and $baaaba$, where the output $+$ means 'yes' and $-$ means 'no'. From these two executions we can infer that the words

$$\epsilon, ab, abaa, ba, baaa$$

are in the language accepted by this FA, and the words

$$a, aba, abaab, abaabb, b, baa, baaab, baaaba$$

are not in that language.

Motivation The FA generalizes a function that just maps each input symbol to a boolean output value. The output of a FA is not just a function of the current input symbol, but may also depend on the preceding input symbols. But a FA can only distinguish a finite number of classes of preceding symbols. The finite memory of the past limits the computational power (§2.6) severely. Considering only yes/no output values may seem to be a limiting factor as well, but that is not the case (cf. decision problem in §2.2).

[3]The automaton is called incomplete or partial (as opposed to complete or total) if one or more combinations of state and input symbol are missing; it is called multivalent or nondeterministic (as opposed to univalent or deterministic) if there are multiple transitions for the same state and input symbol.

Properties

1. The set of input sequences that lead to an accepting state (that yield output 'yes') is called the **language** (§2.3) accepted (recognized, generated) by the FA.
2. The FA is said to solve the decision problem corresponding to its language.
3. Two different FAs can accept the same language, in which case they are called **equivalent FAs**.
4. There exist languages that are not accepted by any FA. An example is the language of **marked palindromes**, having the form $sm\tilde{s}$, where s is an arbitrary word, m is a special marker symbol, and $\tilde{s}$ is the reverse of s. In other words, the decision problem of determining whether a word is a marked palindrome cannot be solved by a FA.

Applications FAs (including cellular automata) are applied in many fields, including electrical engineering, linguistics, philosophy, biology, chemistry, mathematics, ICT.

Questions

- Is a given input sequence accepted by a given FA?
- Design a FA that accepts a given language, i.e., solves a given decision problem on words.
- What language does a given FA accept? The FA can be given in textual or graphical format, but also through an implementation that you can subject to experiments (black box).
- Are two given FAs equivalent?

Extra Terminology variants: finite automaton (FA), finite-state automaton (FSA), finite-state machine (FSM), state machine (SM).

Definition variants: also see Wikipedia below.

- **Nondeterministic** finite automata (NDFA) allow multiple initial states, multiple transitions involving the same state and input symbol, and epsilon transitions (that do not consume an input symbol).
- A **Moore machine** produce an output symbol from a given output alphabet in each state (rather than just a yes/no).
- A **Mealy machine** has transitions (rather than states) that specify an output symbol from a given output alphabet.

The Unified Modeling Language (UML) has a diagram type for defining State Machines, that generalizes the notion of FA as defined here.

Wikipedia Finite automaton — also browse Cellular automaton

2.5 Reduction

Definition A **reduction** is a transformation of one entity (a computational problem, a computation such as of a FA) into another entity of the same kind, while preserving some property. In case of computational problems, we are interested in reductions that preserve the answer, that is, where every instance of one problem can be transformed (reduced) to an instance of another problem, such that the answer to the second instance can be transformed back into an answer for the first instance. In case of computations, we are interested in reductions that preserve the solved problem (accepted language).

Purpose A typical use of reductions is to show that a computation in some model of computation can be transformed into an equivalent computation in a simpler model of computation. This explains the name 'reduction'. Reductions are used to show that one problem/model of computation is no harder/more powerful than another (see §2.6 on computational power).

Wikipedia Reduction (complexity)

2.6 Computational Power (of a Model of Computation)

Definition The **computational power** of a model of computation can be measured by the collection of all languages that can be accepted (the collection of all decision problems on words that can be solved) by computations in that model of computation. The more languages can be accepted (the more decision problems can be solved), the more powerful the model.

Properties The computational power of FAs is the same as that of many FA variants. One such variant is where the output is not determined by whether the current state is an accepting state, but where the output is specified with each transition (including the initialization). The latter variant is known as a **Mealy machine**, and the former as a **Moore machine**. Every Moore machine can easily be reduced (§2.5) to an equivalent Mealy machine: take as output of a transition in the Mealy machine whether the next state in the Moore machine is accepting. The converse is also true, but that reduction is a bit harder, because every Mealy state may have to be split into two Moore states, one accepting and the other not accepting: a Mealy transition with output b is transformed into a Moore transition to the state copy that corresponds to output b.

Extra This meaning of computational power is not to be confused with a performance indicator for computer equipment, such as a CPU, whose operating speed on floating-point numbers is expressed in number of floating-point operations per second (FLOPS).

2.7 Robustness (of a Model of Computation)

Definition A model of computation is said to be **robust**, when small changes in the definition of that model do not affect its computational power (§2.6). That is, the variants are equivalent in computational power. This is typically shown by reductions (§2.5).

Purpose Robustness shows that certain boundaries between computational powers are independent of some technical details in the definition of the relevant model of computation. Examples are the computational power of finite automata (§2.4), and of Turing machines (§2.9).

2.8 Regular Expression

Definition A regular expression (RE) is a specific kind of formula that defines a language (§2.3). REs are constructed from constants and operators. The constants are **0** (denoting the empty language $\emptyset$), **1** (denoting the language $\{\varepsilon\}$ consisting of the empty word), and every symbol x (denoting the language consisting of just the one-symbol word x). The operators are **concatenation** (a binary operator written as juxtaposition, denoting language concatenation), **sum** (a binary operator written as $+$, denoting language union), and **iteration** (a unary postfix operator written as a superscript *, also known as the Kleene star or Kleene closure, denoting zero or more repetitions). Parentheses are used for grouping.

A language that can be defined by an RE is called a **regular language**.

Properties

1. Two different REs can define the same language, in which case they are called **equivalent REs**.
2. **Kleene's Theorem**: A language can be accepted by a FA if and only if it can be defined by a RE. Alternative phrasing: A decision problem can be solved by an FA if and only if it can be specified by an RE.
3. Consequently, not all languages are regular. In particular, the language of marked palindromes (see §2.4, Properties) is not regular.

Applications REs serve to specify decision problems on words.

Many software tools (such as text editors) can perform operations on the basis of a regular expression, such as searching for a pattern in a text. REs are also used to prescribe the expected format of character sequences, such as email addresses, dates, variable names, credit card numbers, etc.

Questions

- Does a given word match a given RE?
- Define a RE that defines a given language.
- What language does a given RE define?
- Are two given REs equivalent?

Extra Terminology variants: regular expression (RE, regexp), pattern

The operators appear with different names and notations: alternation (for sum), $|$ (for $+$), repetition (for iteration).

Definition variants, also see Wikipedia below. For practical application, often more operators are allowed, such as a question mark (?) for an option (zero or one of the preceding), and superscript $^+$ for one ore more repetitions.

Wikipedia Regular expression

2.9 Turing Machine

Definition A **Turing machine** (TM) is a simple computational mechanism with, what is believed to be, universal computational power. It generalizes the finite automaton (§2.4), retaining some FA features.

Input is offered on a two-way unbounded[4] **tape** divided into cells (squares). Each cell is either **blank** (denoted by □), or contains a (non-blank) **symbol** from a finite tape **alphabet**. Output is delivered on that same tape. The tape is accessed by a movable **read/write head**, or (tape) **head** for short. The head scans the symbol on the tape cell under the head, and optionally can modify it.

At the start, the tape typically contains a finite number of consecutive non-blank symbols, with he head positioned over the leftmost non-blank symbol (if any; otherwise, on any blank).

During **execution**, the TM is in one of a finite set of **states**, starting in the given **initial state**. One cannot directly observe the current state of a TM; it is a **black box**. The TM's execution is driven by a **transition function**, that determines for each combination of current state and scanned tape symbol, (a) what symbol will be written on the tape, (b) whether the head will move one cell left or right, and (c) what the next state will be. The combination of a current state, scanned tape symbol, written symbol, head direction, and corresponding next state is known as a **transition**. There can also be special **halting states**, for which the transition function gives no next state. When the TM enters a halting state, its execution halts, and the resulting word on the tape is then considered to be the output. Note, however, that the execution of a TM need not halt on some, or even all, inputs.

Formally, a TM is described by the following items.

1. A finite set, whose elements are referred to as *states*, serving as identifiers for the TM's states
2. One state marked to be the *initial state*
3. Some states marked to be *halting states*
4. A finite alphabet, whose symbols are referred to as *tape symbols*
5. A function, referred to as *transition function*, that, for every possible combination of non-halting current state and scanned tape symbol, determines
   (a) the new tape symbol, to be written on the position of the scanned tape symbol, thereby replacing the current tape symbol
   (b) the direction (L for left or R for right) of movement of the tape head, and
   (c) the next state.

Diagrams for TMs are like diagrams for FAs, but now transitions are labeled with the input symbol, written symbol, and head direction. Halting states are shown as a double circle, and they have no outgoing transitions. A non-halting state can be labeled with a direction (L or R) as an abbreviation that for all tape symbols not being a scanned symbol in an outgoing transition, the TM writes that same symbol, moves the tape head in the indicated direction, and remains in the current state.

Consider the TM defined by the table in Figure 2.91. It has three states (one of which is halting), tape alphabet $\{ 0, 1 \}$, and it flips each bit, leaving the head again on the leftmost bit. Informally, its behavior can be described as follows.

[4] The tape can be extended indefinitely in both directions, as needed; you may view it as infinite.

state	marked as	$\square$	0	1
$q0$	initial	$\square\ L\ q1$	$1\ R\ q0$	$0\ R\ q0$
$q1$		$\square\ R\ q2$	$0\ L\ q1$	$1\ L\ q1$
$q2$	halting			

Figure 2.91: A table defining a 3-state TM that flips all bits and halts at the left

1. Move the tape head to the right while flipping each bit ($0 \leftrightarrow 1$), until reaching the first blank (cf. state $q0$).
2. Then, move the tape head to the left without changing the tape, until reaching the first blank (cf. state $q1$).
3. Move the tape head one step to the right and halt (cf. state $q2$).

The diagram in Figure 2.92 defines the same automaton as the table in Figure 2.91. In this diagram,

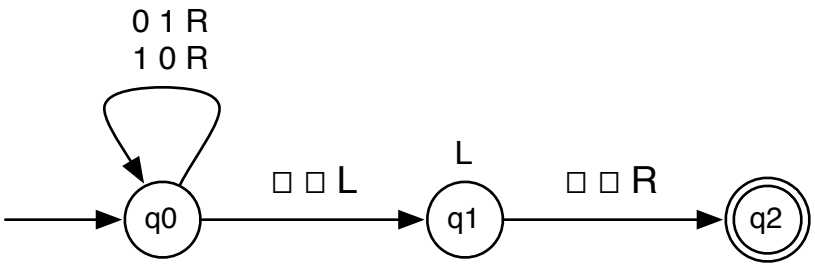

Figure 2.92: A diagram defining a 3-state TM that flips all bits and halts at the left

you see two abbreviations. For state $q0$, the two outgoing transitions with scanned symbols 0 and 1 are shown with a single arrow. State $q1$ has a label L above it, implying outgoing transitions labeled $0\ 0\ L$ and $1\ 1\ L$ to state $q0$ itself.

Turing machines in this course always (must) satisfy the following two constraints[5].

- There exists *exactly one* initial state.
- For each combination of non-halting current state and tape symbol, there exists *exactly one* transition.

Formally, an **execution** or **computation** of a TM for a given initial tape content consists of the sequence of states that the TM traverses while operating on that tape together with the corresponding tape contents and the position of the tape head. Figure 2.93 shows the execution of the TM defined in Figure 2.92 with 10 as the initial tape content. This execution terminates and the TM halts, leaving 01 as final tape content.

History TMs were invented by Alan Turing around 1936 as a tool to help prove that certain mathematical problems admit no 'mechanical solution'.

[5]These are so-called *deterministic* TMs, as opposed to *non-deterministic* TMs that have some behavioral freedom.

tape with head	state	transition label
$\cdots$ □ 1 0 □ $\cdots$	$q0$	1 0 R
$\cdots$ □ 0 0 □ $\cdots$	$q0$	0 1 R
$\cdots$ □ 0 1 □ $\cdots$	$q0$	□ □ L
$\cdots$ □ 0 1 □ $\cdots$	$q1$	1 1 L
$\cdots$ □ 0 1 □ $\cdots$	$q1$	0 0 L
$\cdots$ □ 0 1 □ $\cdots$	$q1$	□ □ R
$\cdots$ □ 0 1 □ $\cdots$	$q2$	

Figure 2.93: An execution of the TM defined in Figure 2.92 with 10 on the tape

Properties

1. The set of input sequences that lead to a halting state is called the **language** (§2.3) accepted (recognized, generated) by the TM. Two different TMs can accept the same language, in which case they are called **equivalent TMs**.
2. Every language that can be accepted by a FA can be accepted by a suitable TM. In fact, every FA can easily be reduced (§2.5) to an equivalent TM. Thus, the computational power (2.6) of TMs is at least that of FAs.
3. There exists a TM that accepts exactly all marked palindromes, which is not the language of any FM. Thus, the computational power of TMs exceeds that of FAs.

Questions

- What is the output tape for a given TM with a given input tape?
- What language does a given TM accept?
- Define a TM that accepts a given language.

Extra Definition variants:

- Additional tape head operations, e.g., not moving;
- Multiple tapes and tape heads

Wikipedia Turing machine

2.10 Church–Turing Thesis

Definition The **Church–Turing Thesis** is the conjecture/hypothesis that a function can be effectively computed if and only if it is computable by a TM. Although the Church–Turing Thesis is in principle unprovable, many, very different, computational mechanisms turn out to have the same computational power as TMs.

Extra Terminology variants: see Wikipedia.

Wikipedia Church–Turing Thesis

2.11 Universal Turing Machine

Definition A **Universal Turing Machine** (UTM) is a specific Turing machine (§2.9) taking as input the **encoding** of any TM $\mathcal{T}$ and the contents of any input tape $\mathcal{I}$ for that TM. Execution of the UTM on this encoding produces the same output as operation of $\mathcal{T}$ would produce on input tape $\mathcal{I}$. Thus, a UTM is **programmable**, and can behave like (can simulate) any other TM.

An encoding of a TM for the UTM is also called a **program**.

Properties

1. What this UTM also shows is that computations can be manipulated as data: the UTM takes as input a description of a computation (a program), and it can execute (interpret) that program. Or, to put it the other way round, data can also encode something that when properly interpreted results in (computational) behavior. Thus, data and programs are interchangeable, or, information and computation are inseparable.

Extra Terminology variants: universal Turing machine, universal machine

Related to: stored-program computer

Wikipedia Universal Turing machine

2.12 Random-Access Machine

Definition A **Random-Access Machine** (RAM) has a memory consisting of a sequence of addressable **registers**, where each register contains a natural number. Each register is identifiable by a unique **address**, which is a natural number. By 'register r' we mean the register whose address is r.

The computation of a RAM is controlled by a program consisting of a sequence of **instructions**. During execution, there is an **instruction pointer** IP that points to the next instruction to be executed. These instructions typically include:

1. $CLR(r)$: (Clear) Set content of register r to 0; increment IP by 1
2. $INC(r)$: (Increment) Add 1 to content of register r; increment IP by 1
3. $JZ(r, z)$: (Conditional Jump) If the content of register r equals 0, then set IP to z; otherwise, increment IP by 1
4. $ILD(r, s)$ (Indirect load) Copy the content of the register whose address is in register s to register r; increment IP by 1
5. $IST(r, s)$ (Indirect store) Copy the content of register r to the register whose address is in register s; increment IP by 1

6. H: Halt

Accessing a register is a unit-time operation. This motivates the name 'random access': a random register can be accessed in a fixed amount of time. Note that the amount of a time a TM needs to access a particular tape cell depends on the distance of the tape head to that cell, which is known as **sequential access**.

Properties

1. RAMs have the same computational power as TMs, but in general they are easier to program.

Extra Terminology variants: instruction, statement, operation

Wikipedia Random access machine

3 Algorithms

Definition Briefly stated, an **algorithm** is a program for an (abstract, idealized) random-access machine (§2.12) that solves a specific computational problem.

For this theme, we refer to [2, Ch.1, 2, 3, 5, 6].

Instead of addressing registers via natural numbers, we will use a finite set of named **variables** that are either of a primitive type (typically integer), or of an **array** type. Arrays are **indexable** by an integer expression, whose value lies in a range compatible with the array's length. Array elements are of a primitive type or an array type. Indexing an array is a unit-time operation.

Algorithms are expressed using the following **statements**.

1. **Assignment**: evaluate an **expression** possibly involving variables, and assign the resulting value to a variable or array element;
2. **Statement block**: a sequence of statements, to be executed in the given order;
3. **Selection** (conditional, if-statement): select among two statements on the basis of a boolean condition;
4. **Bounded repetition** (for-loop): repeat a statement a predetermined number of times. The number of iterations can vary from execution to execution of the for-loop, but it is determined before the loop starts.
5. **Unbounded repetition** (while-loop): repeat a statement on the basis of a boolean condition.

Algorithms should not be confused with **computer programs** that are written in a (concrete) **programming language** for a (physical) **computer**.

Wikipedia Algorithm — Computer program — Programming language

3.1 Specification of Algorithms

Function to be computed; inputs, outputs, and intended relationship; cf. §2.1 about Computational Problem

3.2 Description of Algorithms

Name, input (possibly with a precondition), output, goal (intended relation between input and output), the steps in terms of input/output parameters, local variables, and statements. Cost (run-time, memory usage).

Statements:

- Return the value of an expression as result;
- Set a variable value from an expression; including setting the value of an element in an array, where the element is selected through an index expression;
- Conditional execution, using if-then-else statement;

- Repetition, using for- or while-statement;
- Invoking a function with given actual parameters; including the function being defined (recursion).

Functions can have local variables.
If a statement in an algorithm invokes that algorithm itself, then this is called **recursion**.

Wikipedia Variable (computer science) — Statement (computer science) — Control flow — Recursion (computer science)

3.3 Execution of Algorithms

Extra Also see `http://visualgo.net/`

3.4 Correctness of Algorithms

Definition The **correctness** of an algorithm is expressed in relation to a specification (§3.4). An algorithm is said to be correct for a given specification, when every execution (3.3) of the algorithm, starting with an input that satisfies the specified precondition, will terminate such that the output establishes the specified goal.
A **loop invariant** is . . . initialization, invariance, finalization; termination

Wikipedia Loop invariant — Mathematical induction

3.5 Runtime of Algorithms

Definition The **runtime** of an algorithm on a specific input is measured by counting basic steps of the algorithms execution on that input, where array indexing takes unit time, The **runtime complexity** of an algorithm is the algorithm's runtime as function of input size.
The **asymptotic runtime complexity**; order of magnitude notation O, Ω, Θ.

Wikipedia Big O notation

3.6 Standard Algorithms

See [2, Ch.5, 6]:

- On sequences:
  - Searching: Linear, Binary
  - Sorting: Selection, Insertion, Merge, QuickSort
- On graphs:
  - Topological sorting
  - Dijkstra's shortest path algorithm

4 Information

Definition **Information** is what reduces, or even completely takes away, **uncertainty** in the receiver. It provides (at least a partial) answer to a question. The uncertainty exists because multiple answers are possible, and the receiver does not know which one applies. If only one answer is possible, then there is no uncertainty.

An answer can be conveyed through a **symbol**, or even a sequence of symbols, whose meaning has been agreed upon beforehand. For decision problems (§2.2), two symbols suffice to encode the answer: one symbol, say 1, to denote 'yes', and the other, say 0, for 'no'.

In 1948 [7], Shannon stated

> "The fundamental problem of communication is that of reproducing at one point either exactly or approximately a message selected at another point."

Similarly, the fundamental problem of information storage is that of reproducing at one time either exactly or approximately a message selected at an earlier time. Thus, it is the selection that conveys information, not the message or symbol itself.

Shannon continued:

> "Frequently the messages have meaning; that is they refer to or are correlated according to some system with certain physical or conceptual entities. These semantic aspects of communication are irrelevant to the engineering problem. The significant aspect is that the actual message is one selected from a set of possible messages. The system must be designed to operate for each possible selection, not just the one which will actually be chosen since this is unknown at the time of design."

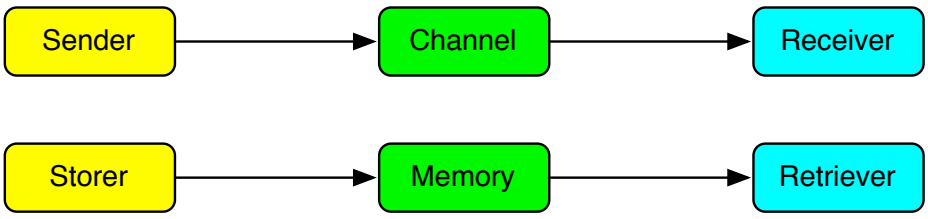

Figure 4.01: Concepts involved in communication and storage of information

Problems studied in this course:

- **Efficient** communication and storage
- **Reliable** communication and storage, in the presence of random errors (noise)
- **Secure** communication and storage, in the presence of adversaries

Note that these problems of communication and storage do not, as such, involve computations on the information. The information is to be transferred across space or time unchanged. However, solutions to these problems heavily rely on computations.

Extra Khan Academy on YouTube: Language of Coins (Information Theory), especially 1, 4, 9, 10, 12–15.

Terminology variants: information theory, coding theory

Wikipedia Information — Information theory — Coding theory

4.1 Efficient Communication and Storage

Efficiency of communication and storage can be measured in terms of the length of the symbol sequences needed to encode messages. This length directly correlates to amount of energy, (communication) time and (storage) space needed, and hence to cost. More efficient codes use shorter symbol sequences. This topic is studied under idealized circumstances where the medium is perfect (introduces no errors). Furthermore, we make the assumption that the communication or storage cost per symbol is constant, that is, all symbols cost the same (in terms of energy, time, space).

4.2 Information Source

Definition An **information source** produces a sequence of **messages** (answers, symbols), taken from a given **alphabet** (set of symbols). An information source serves as a *model* of a class of senders, so that we can study techniques to improve communication for such senders.

An important class of information sources is *stochastic* in nature, that is, the source produces a sequence of messages according to some probability distribution. An information source with finite alphabet, whose messages are *independent and identically distributed*, is known as a **discrete memoryless source**. It is called memoryless, because the probability distribution for each symbol produced as output is independent of previous output symbols.

Let S be an information source with alphabet A. By considering blocks of n adjacent source symbols as new symbols (known as **block symbols**), one obtains a new source, known as **block source** denoted by $S^{(n)}$, with alphabet A^n. The probability distribution of $S^{(n)}$ is completely determined by that of S.

Extra A **Markov information source**, or **Markov source** for short, is an information source with finite (bounded) memory, like a finite automaton (§2.4). More formally, at any moment during the production of symbols, a Markov source is in one of a finite set of **states**. The state transitions and outputs are determined by a **transition table** that gives the probability to go from state i to state j producing symbol a (a kind of probabilistic Mealy machine). Alternatively, one can say that the probability distribution of the output depends on the preceding k output symbols, for some finite number k. This is known as a Markov source order-k. A discrete memoryless source is a Markov source of order 0.

Markov sources can be used to approximate the distribution of symbols occurring in various natural information sources, such as natural language texts. This is used in predictive keyboards. The higher the order, the better the approximation.

Wikipedia Information source (mathematics) — Markov information source

4.3 Codes: Encoding and Decoding

Definition An **encoding** is a function that maps an input sequence of symbols to an output sequence of symbols, possibly using a different alphabet.

Figure 4.31: Placement of encoder and decoder

An encoding is **lossless** when the input can be uniquely recovered from the output, using a **decoding** function that acts as an inverse of the encoding.

A **code** consists of an encoding and a corresponding decoding.

A *sender together with an encoder* can be viewed as a new information source (§4.2), with transformed properties, such as a more uniform probability distribution. A *channel together with an encoder and a decoder* can be viewed as a new channel, with transformed properties, such as being more efficient than the plain channel (without encoder–decoder).

There are various types of encodings for symbol sequences (streams):

Block codes first cut the input sequence up into blocks, and encode each input block separately as an output block. Block lengths of input blocks and of output blocks can vary (see below). The output blocks are also known as **code blocks** or **code words**.

Block codes contrast with **convolutional codes** that do not work by cutting up the input sequence. Rather, they work like a (finite) state machine, consuming input symbols one by one, and producing output symbols based on the input and the current state. Convolutional codes are outside the scope of this course.

Fixed-length codes are block codes where all input blocks are equal in size, and also all output blocks are equal in size. Input block size and output block size need not be the same.

Variable-length codes are (usually) block codes with a fixed input block length and variable output block length; that is, the length of each output block depends on the input block.

Alternatively (less common), the input block length is variable (according to some algorithm), and each input block is encoded separately as an output block (either of fixed-length, for a variable-length-to-fixed-length code; or of variable-length, for a variable-length-to-variable-length code).

Prefix-free codes[6] are a special class of variable-length codes, where no code block is a **prefix** of another code block, that is, an initial segment of a code block is never itself a code block.

The **code rate**[7] or **compression ratio**[8] of an encoding is the (average) number of the input symbols encoded per output symbol. If the input and output alphabets are the same (often, they are bits) then

[6] Sometimes confusingly also referred to as *prefix codes*.
[7] This terminology is mainly used in the communication community.
[8] This terminology is mainly used in the storage community.

the rate is a dimensionless scalar. It is a measure of the encoding's **efficiency**. A higher compression ratio or code rate corresponds to a more efficient code. When the rate is less than 1, we speak of **inflation**, rather than compression. Inflation is useful to improve reliability (cf. §4.8–§4.13).

Properties

1. Fixed-length codes are a special kind of prefix-free codes.
2. Variable length output blocks create a concern for **unique decodability** of the output sequence. In the case of a fixed-length code, you can simply count off the required number of symbols in the output stream to reconstruct the output blocks. Note that in a variable-length code there are no spaces or commas to separate the code words and help recognize them in the output stream.
3. Prefix-free codes (hence, also fixed-length codes) allow **unique decoding**, provided that no two distinct input blocks encode to the same output block.
4. A prefix-free code can conveniently be represented in a **tree**, where all prefixes of codewords serve as **nodes**, including the empty word that serves as root node of the tree, and all codewords that appear as leaf nodes. For every symbol a, and sequence s, such that sa is a prefix of a codeword, there is an **edge** with label a from the node for s to the node for sa. This tree can be used for encoding and decoding. In fact, you get a finite automaton (§2.4) that recognizes the codewords, by combining the root node and leaf nodes into one node, which serves as initial and only accepting state.
5. A fixed-length code with input blocks of length m and output blocks of length n has a rate or compression ratio of m/n input symbols per output symbol.
6. Variable-length codes (including prefix-free codes) are very sensitive to noise. An error in a single output symbol has a **ripple effect** when decoding the output sequence, because it may affect the size of blocks being reconstructed.

Applications Some well-known data compression algorithms are (see [2, Ch.9]):

Huffman's algorithm (also see §4.4) constructs an prefix-free encoding that is optimal for a given information source;

Run-Length Encoding (RLE) is a variable-length-to-variable-length encoding that basically encodes each **run** of equal symbols by its length (and some indication of the symbol);

Lempel–Ziv–Welch (LZW) algorithm is a good (but not truly) **universal** compression algorithm that does not require prior knowledge of source statistics. It is a variable-length encoding. LZW is used in popular data compression programs such as ZIP.

Extra Terminology variants: Prefix-free codes are also known as **instantaneous** codes, because they can be decoded instantaneously, that is, as soon as a complete codeword has been received.

Wikipedia Code — Convolutional code — Data compression ratio — Code rate

4.4 Huffman's Algorithm

Definition **Huffman's algorithm** [2, Ch.9] takes as input the probability distribution of a discrete memoryless information source (§4.2), and produces as output an encoding-decoding tree (§4.3, Property 4) for an optimal prefix-free compression code (§4.3).

Huffman's algorithm works as follows by combining messages into 'super'messages:

1. It starts with an empty encoding, where all messages are separate nodes, forming a so-called **forest** of one-node trees.
2. In case there is (remains) only *one* (super)message, there is no (further) need to extend the encoding, because the forest then consists of a single tree, and the algorithm terminates by returning this tree.

   In particular, if the source only emits one type of message, then its probability equals 1 and there is no need to encode anything (one choice equals no choice).
3. In case there is more than one (super)message, Huffman's algorithm combines the trees for the *two* messages with *least* probability (highest information content), say A_0 and A_1, and distinguishes between them using one bit, say 0 for A_0 and 1 for A_1 (see Fig. 4.41).

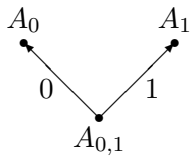

Figure 4.41: Huffman's algorithm combines two trees A_0 and A_1 into one tree $A_{0,1}$.

4. Now the algorithm is repeated (recursively) from Step 2 for a new source where messages A_0 and A_1 have been merged into one new supermessage $A_{0,1}$ with probability $P(A_{0,1}) = P(A_0) + P(A_1)$. So, this new source has one fewer message (the forest has one fewer tree). In the resulting encoding for this new source, the encoding of $A_{0,1}$ is extended with one bit to obtain encodings for A_0 and A_1.

Properties

1. Of all possible prefix-free codes to encode the given source, Huffman's algorithm produces one that is *most efficient* (optimal), in the sense of using, on average, the least number of bits per source symbol. In other words, there exist no prefix-free codes that are more efficient than the one produced by Huffman's algorithm.
2. Huffman's algorithm as described above is *nondeterministic*, in the sense that its steps leave some implementation freedom:
   - In Step 3, there could be multiple choices for the two messages with least probability. More precisely, this happens when the third least probability equals the second least probability. Which two are selected does affect the produced encoding, but not its efficiency.
   - In Step 3, the choice of which message to encode by 0 and which by 1 does affect the produced encoding, but not its efficiency,

As a consequence, applying Huffman's algorithm multiple times (e.g. by different persons) to the same information source, may result in different (optimal) encodings.

3. There can be multiple optimal codes (see Property 2). In fact, there exist information sources that have an optimal encoding which is not the result of any implementation of Huffman's algorithm.

4. Huffman's algorithm is a **greedy algorithm**. It can be implemented efficiently with a worst-case running time of $\mathcal{O}(n \log n)$, where n is the number of source symbols. This is done by using a **priority queue** implemented as **heap**, making it possible both to find a message with minimum probability and to update the data structure when merging two messages in $\mathcal{O}(\log n)$ time.

5. The resulting encoding and decoding algorithms are also efficient in running time ($\mathcal{O}(k)$ to encode/decode a message that is encoded in k bits) and required storage ($\mathcal{O}(n)$ to store the encoding tree).

6. If the compression ratio resulting from Huffman's algorithm is not satisfactory, one can consider the *block source* (§4.2) that emits blocks of n symbols from the original source, and again apply Huffman's algorithm to this block source. As n increases, the resulting compression ratio can improve, but the improvements diminish as n grows. Note that there is a hard upper bound to the achievable compression ratio (cf. §4.7).

7. Huffman's algorithm is not **universal**. It is optimal only for the specific probability distribution of the *memoryless* source for which the encoding is constructed. Moreover, the encoding tree needs to be known by the receiver. Typically, it is built into the receiver, or communicated separately through another channel. Also see the variants below.

Variants There are several variants of Huffman's algorithm:

n-ary Huffman coding Huffman's algorithm is easily generalized to channels that are not binary, but work with three or more symbols. Simply replace all occurrences of 'two' in Step 2 by the number of available channel symbols.

Two-pass Huffman coding Requires buffering of the sequence to be encoded. In the first pass through the sequence, the symbol frequencies are determined to estimate the probabilities. In the second pass, the actual encoding is constructed. Afterwards, both the encoding tree and the encoded sequence need to be communicated.

Adaptive Huffman coding It is also possible to maintain a dynamic encoding tree that is adjusted while source symbols are encoded. Changes in the encoding tree must now also be communicated to the receiver.

Wikipedia Huffman coding

4.5 Information Measure

Definition The **amount of information** $\mathcal{I}(A)$ conveyed by message A is defined by

$$\mathcal{I}_r(A) \;=\; -\log_r P(A) \tag{4.51}$$

where $P(A)$ is the probability of message A and $\log_r$ the logarithm to base r. The choice of base r concerns a scaling factor, since

$$\log_r x \quad = \quad (\log_r s) \log_s x \tag{4.52}$$

The **unit of information** called **bit**[9] corresponds to base $r = 2$. Thus, 1 bit is the amount of information conveyed by a message with a probability of 0.5, since

$$-\log_2 0.5 \quad = \quad 1$$

Alternatively, one bit is the amount of information you get, when receiving an answer to a binary question with two equiprobable answers (like the outcome of the flip of a fair coin).

Properties The following properties are independent of the chosen unit of information, that is, of the scaling factor determined by base r. Therefore, we left out the subscript r.

1. $\mathcal{I}(A) \to \infty$, if $P(A) \to 0$ (an impossible message never occurs)
2. $\mathcal{I}(A) = 0$ (no information), if $P(A) = 1$ (certainty): $-\log 1 = 0$
3. $0 \leq \mathcal{I}(A) < \infty$, if $0 < P(A) \leq 1$
4. $\mathcal{I}(A)$ is a **decreasing** function of $P(A)$: a lower probability corresponds to a higher amount of information, that is, $\mathcal{I}(A) > \mathcal{I}(B)$, if and only if $P(A) < P(B)$
5. $\mathcal{I}(A)$ is a **continuous** function of $P(A)$: a small change in $P(A)$ causes a small change in $\mathcal{I}(A)$
6. $\mathcal{I}(AB) \leq \mathcal{I}(A) + \mathcal{I}(B)$: information is **subadditive** (AB stands for receiving messages A and B)
7. $\mathcal{I}(AB) = \mathcal{I}(A) + \mathcal{I}(B)$ (information is **additive**, **if** A and B are statistically independent)

The desire to have that last (additive) property of $\mathcal{I}$ motivates the appearance of the logarithm in the definition of $\mathcal{I}$. The probability of answer AB to a composite question, where the answers to the two components are independent, satisfies:

$$P(AB) \quad = \quad P(A)P(B) \tag{4.53}$$

and hence $\mathcal{I}(AB) = -\log P(A)P(B) = -\log P(A) - \log P(B) = \mathcal{I}(A) + \mathcal{I}(B)$.

Extra Properties 4, 5, and 7 uniquely determine the definition of $\mathcal{I}$, apart from a scaling factor.
Shannon's approach to information involves stochastic information sources based on a probability distribution, and is also known as **Probabilistic Information Theory** (PIT). It contrasts with another approach named **Algorithmic Information Theory** (AIT). PIT makes statements about averages taken over many sequences, whereas AIT also applies to individual sequences. For instance, AIT defines the (algorithmic) information content of a sequence s as the length of a shortest program that produces s as output. Asymptotically (for long sequences) the PIT and AIT information measures agree. A decoder for a code can be viewed as a computational machine, and its encoded input as a program that produces the decoded output. In general, however, this machine is not universal. One could say that in AIT, a generic universal 'decoder' is used, and messages are encoded as programs for that machine, which executes the program to produce the decoded message as its output.
Terminology variants: information measure, information content, self-information, surprisal

[9] Bit is a contraction of **binary digit**.

Wikipedia Self-information — Algorithmic information theory

4.6 Entropy

Definitions The **entropy** $\mathcal{H}(S)$ of an information source S is the expected (mean) amount of information per symbol, taken over all sequences of symbols producible by S.

For a discrete memoryless information source, the entropy measured in bits is given by the weighted average:

$$\begin{aligned} \mathcal{H}(S) &= \sum_{A \in S} P(A)\mathcal{I}_2(A) \\ &= -\sum_{A \in S} P(A)\log_2 P(A) \end{aligned} \tag{4.61}$$

The relevance of entropy becomes clearer in Shannon's Source Coding Theorem (see §4.7) and Channel Coding Theorem (see §4.13).

The difference $\mu - \mathcal{H}$ between the actual (average) number μ of symbols per message and the entropy $\mathcal{H}$ (which turns out to be the theoretical lower bound according to Shannon's Source Coding Theorem in §4.7) is called **(absolute) redundancy**. It is the amount that is superfluous when conveying the information. The **relative redundancy** is the ratio $(\mu - \mathcal{H})/\mathcal{H}$, which can also be expressed as a percentage.

Properties For a discrete memoryless source S, we have:

1. **Entropy bounds**: $0 \leq \mathcal{H}(S) \leq \log_2 N$, for a source with N messages
2. $\mathcal{H}(S) = 0$, if and only if $P(A) = 1$ for some message $A \in S$ (certainty)
3. $\mathcal{H}(S) = \log_2 N$, if and only if $P(A) = \frac{1}{N}$ for all A (maximum uncertainty)
4. $\mathcal{H}(S^{(n)}) = n\mathcal{H}(S)$ for all $n \geq 1$ (see §4.2 for block source $S^{(n)}$)
5. The entropy of a binary source S with $P(1) = p$ and $P(0) = 1 - p$ is given by the **binary entropy function** $\mathcal{H}(p)$ defined for $0 \leq p \leq 1$ by (also see Fig. 4.61):

$$\mathcal{H}(p) = -p\log_2 p - (1-p)\log_2(1-p) \tag{4.62}$$

Wikipedia Entropy (information theory) — Binary entropy function

4.7 Source Coding Theorem, Data Compression

Theorem Shannon's **Source Coding Theorem** (1948) provides a bound on how much the output of an information source can be compressed without loss, that is, a bound on how much efficiency can be improved without loss:

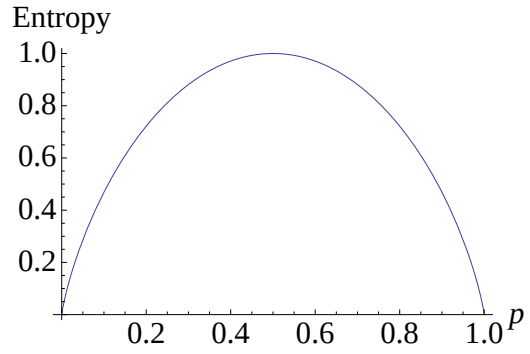

Figure 4.61: The binary entropy function $\mathcal{H}(p)$

On average, each message can be encoded in $\approx \mathcal{H}$ bits and not $< \mathcal{H}$ bits, where $\mathcal{H}$ is the entropy of the message source.

More precisely:

Given an information source S with entropy $\mathcal{H}$,

- for every $\varepsilon > 0$, there exist a pair of *lossless encoding-decoding algorithms*, such that on average, they encode the source S in $< \mathcal{H} + \varepsilon$ bits per message, and
- every lossless encoding-decoding algorithm encodes the source S with an average of $\geq \mathcal{H}$ bits per message, that is, no lossless encoding algorithm can achieve $< \mathcal{H}$ bits per message on average.

Notes on Shannon's Source Coding Theorem:

1. The theorem only applies to *lossless* compression. Lossy compression (like JPG and MPG) could achieve fewer than $\mathcal{H}$ bits per message.
2. The theorem provides a precise bound that works in *two directions*: Compression below the entropy is impossible, and the entropy can be approached from above as closely as one wants.
3. When the source sends out bits, the bound on the **compression ratio** (§4.3) is $1/\mathcal{H}$; a lower source entropy allows for a higher compression ratio.
4. The theorem makes a claim about the *average* number of bits per message. It does not promise that the encoding *always* succeeds with $< \mathcal{H}+\varepsilon$ bits for *each* message separately. Thus, $< \mathcal{H}+\varepsilon$ bits per message may only be achieved in the long run, when sending many messages (cf. The Law of Large Numbers).
5. Which compression algorithm will work depends, in general, on the choice of ε. To achieve a better approximation of $\mathcal{H}$, that is, a smaller value of ε, requires a more advanced compression algorithm. This in turn may require that more messages are combined and encoded together, generally incurring more overhead costs per message.
6. In fact, the proof of the theorem is based on increasing the input block length of the encoding (cf. block source in §4.2), to encode multiple messages together. The code words can basically be assigned almost randomly (see below under Extra for some details).

7. Achieving a better approximation of the entropy (smaller ε) may require an increase in communication **latency**. That is, the delivery of a message may get delayed, since it has to wait until enough later messages are available to do the encoding.

8. The theorem does not exclude a lossless compression to *exactly* $\mathcal{H}$ bits per message (on average). But this only happens in very special cases.

9. The theorem motivates the relevance of entropy, viz. as the limit on compression.

   In fact, good compression will encode (on average) each message close to its information content (§4.5), so that more frequent messages have a shorter encoding than less frequent messages.

10. The assumption is that all channel symbols (bits) cost the same (in terms of time, energy, matter, or money). In practice, this need not be the case.

11. The practical challenge is to strike a balance between

    - improved communication efficiency and
    - complexity and efficiency of the encoding-decoding algorithms.

    Ideal are simple and fast encoding-decoding algorithms that achieve a high compression ratio.

12. In practice, the source statistics are not known, and compression algorithms must estimate them while observing the source.

Properties

1. An encoding, whose input and output alphabet are the same, cannot both be lossless and map *every* input sequence to a shorter output sequence. In other words, there is no truly **universal** compression method.

   Reason: The total number of shorter output sequences is strictly less than the number of input sequences of a given length.[10] If an encoding maps every input sequence to a shorter output sequence, then, by the **pigeonhole principle**, there must be two input sequences that map to the same output sequence. Hence, the encoding is not lossless, because that output sequence cannot be uniquely decoded.

2. If an input sequence is compressed twice in succession, then the second compression will not improve efficiency (and in practice, usually it will cause inflation). And if it does improve compression, then this means that the first compression was not so good.

3. After applying a good compression algorithm, the output sequence will look (almost) **random**, in the sense that for any given length all subsequences of that length appear with equal probability. In case of binary output, the entropy of the output sequence will be close to 1.

   If this were not the case, then the bias in statistics could be exploited for further compression.

4. Huffman's algorithm (§4.4) provides an encoding for a memoryless information source that approaches the entropy when applied to the block source $S^{(n)}$ for sufficiently large n. In the limit $n \to \infty$, it converges to the source' entropy.

[10] For an alphabet with $m \geq 2$ symbols, the total number of sequences of length i equals m^i. The total number of non-empty sequences of length $i < n$ equals $\sum_{i=1}^{n-1} m^i = (m^n - m)/(m-1) < m^n$.

Extra The proof of Shannon's Source Coding Theorem involves the following insights. We formulate this in the setting where the input and output alphabet are bits and the source' entropy is $\mathcal{H}$ bit. Consider all 2^n blocks of n input symbols. One can show that among these blocks, there are roughly $2^{n\mathcal{H}}$ blocks that occur with almost equal probability $2^{-n\mathcal{H}}$ (these blocks are known as *typical* blocks), and the remaining blocks (known as *atypical* blocks) occur with a negligible probability. The typical blocks can be uniquely encoded in output blocks of $n\mathcal{H}$ bits, reserving the all-0 block as a prefix for blocks of length $n\mathcal{H}+n$ to encode the atypical blocks. Because atypical blocks occur with negligible probability, the average number of output bits per input bit is determined by the encoding of the typical blocks. Thus, the encoding uses approximately $n\mathcal{H}/n = \mathcal{H}$ bits per input bit. In other words, its rate is $1/\mathcal{H}$.

Wikipedia Source coding theorem — Data compression — Pigeonhole principle

4.8 Reliable Communication and Storage: Protection against Noise

Reliability of communication and storage can be measured in terms of the probability that a message is received or retrieved erroneously. The lower this probability, the higher the reliability. This is studied for error sources that act 'without purpose' (noise), as opposed to adversaries that attempt to interfere with messages 'on purpose'. For the latter, see §4.14.

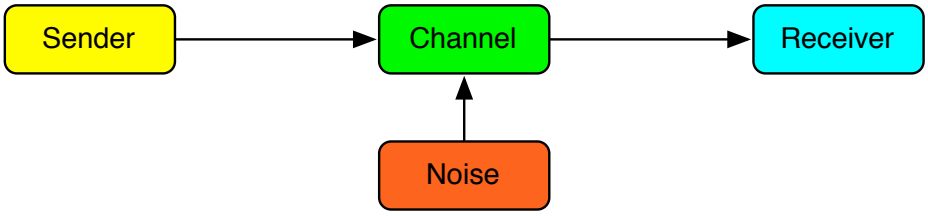

Figure 4.81: Concepts involved in communication involving noise

Note that compression increases the destructive effect of noise, since compression reduces redundancy. To help protect against noise, one needs to encode so as to increase the redundancy, thereby reducing the efficiency. Such codes have a compression ratio (rate) less than 1.

4.9 Noisy Channel Models

Definitions Each physical communication channel and memory is subject to accidental errors, known as **noise**. The specific type of noise to which a channel or memory is susceptible depends on the channel's physical characteristics. There are several important models of noisy channels.

Binary Symmetric Channel (BSC) with bit error probability p transmits bits, where each bit gets transmitted correctly with probability $1-p$, and is in **error**, that is, gets changed to the opposite value ('flipped'), with probability p (see Fig. 4.91). Probability p is therefore called the **(bit) error probability** of the channel.

It is called symmetric, because the error probabilities for 0 and 1 are equal. The bit error for subsequent bits is *independent and identically distributed*, like the discrete memoryless information source (§4.2).

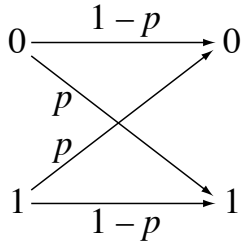

Figure 4.91: Binary symmetric channel with bit error probability p (Wikipedia)

In fact, this type of noise can be viewed as a binary memoryless information source that produces a 1 (to flip the channel bit) with probability p, and a 0 (to pass the channel bit unchanged) with probability $1 - p$. The binary symmetric channel *adds* this noise to the message:

$$channel\ output \quad = \quad channel\ input \oplus noise \tag{4.91}$$

Addition is done **modulo 2**, that is, (like in clock arithmetic, where multiples of 2 are ignored. Addition modulo 2 is denoted by $\oplus$ and satisfies these properties:

$$0 \oplus 0 = 0 \qquad 1 \oplus 0 = 1 \qquad 0 \oplus 1 = 1 \qquad 1 \oplus 1 = 0 \tag{4.92}$$

Thus, when the noise bit is 0, the channel output equals the input, and when the noise bit is 1, the channel output is the opposite of the input:

$$\begin{aligned} x \oplus 0 &= x \\ x \oplus 1 &= 1 - x \end{aligned}$$

Binary Erasure Channel (BEC) with bit error probability p transmits bits, where each bit is communicated correctly with probability $1 - p$, and gets *erased* with probability p, resulting in an **erasure** (see Fig. 4.92). An erasure is recognizably different from a 0 or a 1. But the sender has no control over erasures; hence, it cannot use erasures as a third channel symbol.

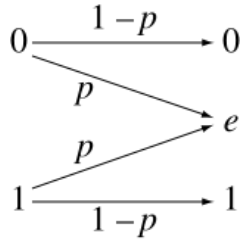

Figure 4.92: Binary erasure channel with bit erasure probability p (Wikipedia)

General Symmetric Channel with an alphabet of k symbols and error probability p transmits symbols, where the symbol is transmitted correctly with probability $1 - p$, and is changed to each of the other symbols with probability $p/(k - 1)$.

Of practical interest is the *decimal* symmetric channel, as a model for human communication of numerical data, such as bank account numbers and product codes.

General Erasure Channel with an alphabet of k symbols and erasure probability p transmits symbols, where each symbol is communicated correctly with probability $1 - p$, and gets **erased** with probability p.

Again, the *decimal* erasure channel is of practical interest. For instance, an illegible character in printed or handwritten text can be modeled as an erasure.

Properties

1. A binary symmetric channel with error probability $p = 1/2$ is practically useless. No matter what bit sequence the sender offers to the channel, it comes out as a completely random sequence when the noise has been added.

   To understand this, suppose that you just received a 1 bit. In 50% of the cases, the original was a 1. In the the other 50%, it was a 0. Similarly, when receiving a 0 bit, the bit sent was a 0 in 50% of the cases and otherwise a 1.

   More formally, this can be formulated using the conditional probability that the bit sent was x when it is given that bit received was y. This conditional probability is denoted by $P(x \mid y)$, and when $p = 1/2$ we have

   $$P(x \mid y) \quad = \quad 1/2$$

   for any combination of x and y. The received bit provides no clue to what was sent. The channel is useless (for communicating information).

2. The general symmetric channel is useless when the error probability equals $(k-1)/k$, because then **Extra**

   $$P(x \mid y) \quad = \quad 1/k, \quad \text{for all } x \text{ and } y.$$

3. A binary symmetric channel with error probability $p > 1/2$ is for all practical purposes equivalent to one with error probability $1 - p < 1/2$. Just flip every bit that is received before further processing.

4. On the binary symmetric channel with bit-error probability p, the probability $P(e)$ of error vector e is given by

   $$P(e) \quad = \quad p^{w(e)}(1-p)^{n-w(e)} \tag{4.93}$$

   where n is the length of e, and $w(e)$, called the **(Hamming) weight** of e, is the number of 1 bits in e:

   $$w(e) \quad = \quad \sum_{i=1}^{n} e_i$$

Extra On a **burst error channel**, subsequent bit errors are not independent, but correlated. Once an error occurred, it is more likely that the next bit is also in error. This is like the Markov information source (§4.2 under Extra). In practice, burst error channels are quite common. For instance, scratches on a DVD are burst errors, because the scratch damages multiple bits located close together. **Interleaving** and **de-interleaving** can be used to protect against burst errors. This works as follows:

- A sequence of k consecutive codewords of length n is *interleaved* by putting them as k rows in a $k \times n$ matrix, and then transmitting the symbols in the matrix column by column.
- The received symbols are de-interleaved by reversing the interleaving process: put the symbols as columns of length k in a $k \times n$ matrix, and then read off the rows as the k received words of length n.

For instance, interleaving the three code words $abcd$, $efgh$, and $ijkl$ results in $aeibfjcgkehl$. A single burst error no longer than k symbols will affect at most one symbol in each of the k consecutive codewords. Interleaving spreads out burst errors, and makes them look more like independent errors.

Wikipedia Binary symmetric channel — Binary erasure channel — Burst error

4.10 Error Control Techniques

Definitions There are two common ways to improve reliability of communication on a noisy channel. Both involve encodings of the messages where only *some* of all possible words (symbol sequences) that could be sent on the channel, are actually used to encode a message. The non-codewords will not be sent by the sender, but they could arise when noise acts on the channel and corrupts symbols (§4.9). The code must be designed specifically to support the chosen technique.

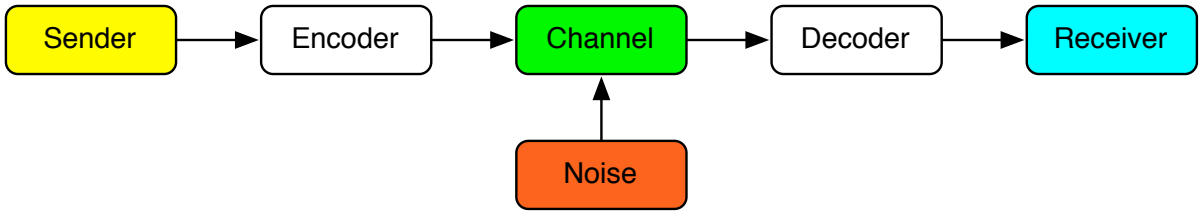

Figure 4.101: Concepts involved in reliable communication

One way to improve reliability in communication[11] is to use an **error-detecting code** and a **feedback channel**. This works as follows.

1. When the receiver receives a word, it first checks whether this word is actually a codeword. If it is, then the receiver assumes that this is the codeword that was sent.

   N.B. If the assumption is not correct, that is, if the received codeword differs from the one sent, then we speak of an **undetectable error**.

2. If the received word is not a codeword, then something must have gone wrong. A transmission error has been *detected*.

[11] This technique is less useful for storage.

3. The receiver then uses the feedback channel to request a **retransmission**. Hopefully, the retransmission works, and otherwise there can be another attempt.

An alternative way to improve reliability in communication and storage is to use an **error-correcting code**. This does not need a feedback channel, and is also known as **forward error correction** (FEC). It works as follows.

1. When the receiver receives a word, it first checks whether this word is actually a codeword. If it is, then the receiver assumes that this is the codeword that was sent.
2. If the received word is not a codeword, then something must have gone wrong. A transmission error has been *detected*.
3. The receiver then determines what was the likely codeword that was originally transmitted, thereby *correcting* the error. This is known as **maximum likelihood decoding**.
4. In some cases, the decoder for an error-correcting code can report that it is incapable of correcting the detected error. This is called an **uncorrectable error**.

After decoding, the result may still be in error, either because the noise changed one codeword into another codeword, and no error was detected, or (only in case of an error-correcting code) because the applied correction was wrong. In those cases, we speak of a **residual error**. The **residual error probability** is the probability that the decoded result is still in error. This probability can be scaled to be comparable to the bit error probability of the channel: **Extra**

1. Consider a binary sequence s of length k (the source bits), the codeword c of length n obtained by encoding s (the code bits sent on the channel), and an **error vector** e (binary sequence of length n, with 1s on the error positions).
2. The receiver receives word $r = c \oplus e$ (where addition $\oplus$ is done bitwise modulo 2), decodes this to codeword d, and extracts k source bits s'.
3. We can then calculate the weight $w(s, e)$ of the residual error as the number of source bits received in error, that is, the number of bit positions where s and s' differ (this is the Hamming distance (§4.11) between s and s'). If that weight equals 0, then there is no residual error.
4. The residual bit-error probability is then the average (mean) taken over all source sequences s of

$$\sum_e P(e)w(s, e)/k \tag{4.101}$$

where $P(e)$ is the probability of error vector e given by (4.93).

Properties

1. Both error-detecting and error-correcting codes work by increasing the redundancy, that is, by using the channel *less* efficiently, so that the effect of noise can be detected, or even corrected.

2. A fixed-length error-detecting code with feedback channel and conditional retransmission is actually a kind of *variable-length code* (§4.3). The number of symbols it takes to transmit a message varies, depending on the number of retransmissions involved. Do note that it involves interaction with the receiver.

3. Error detection is less useful for information storage than for information communication, because there are fewer options for recovery after detecting an error on information retrieved from storage.

4. Error detection with retransmission adds considerable *time overhead*, if communication *latency* is high, as is the case when communicating with far-away space probes. In that case, error-correcting codes are the only reasonable option.

5. Error-detecting codes and error-correcting codes can improve reliability, but neither can provide 100% reliability. Whenever the sender can choose among multiple messages to send, there is the possibility of **undetectable errors**: an unreliable channel could change the encoding of one message into a valid encoding of another message. In that case, the receiver has no clue that something has gone wrong For all that the receiver knows, the sender intended to send that other message. The aim of error control techniques is to make this probability sufficiently small.

Applications

- A **repetition code** repeats every message, that is, it always sends every symbol *twice*. It can serve as an error-detecting code: when the second copy of the symbol is not the same as the first copy, then an error is detected, and a retransmission can be requested.

  Its rate is $1/2$.

- A **parity bit** is a one-bit extension on a binary word to force the number of 1s to be even (and in another variant, to be odd). This can serve as an error-detecting code. When the parity of the received word is not even, it is not a codeword, and an error has been detected. This allows the detection of any single-bit error per codeword.

  If codewords have n bits, then the rate of this code is $(n-1)/n$.

- There are many standardized **decimal error-detecting codes** used in practice, such as:

  - Credit Card Numbers
  - Universal Product Codes (UPC)
  - European Article Numbers (EAN)
  - International Standard Book Numbers (ISBN)
  - International Bank Account Numbers (IBAN)
  - Dutch Citizen Service Numbers (BSN)
  - Student Identification Numbers at TU/e

  These codes use one or more **check digits**, or a **checksum**. They protect against all single-digit errors, but often also against digit transpositions and other common human mistakes.

- A **repetition code** that sends *three* copies of every bit can serve as error-correcting code. If the received symbols are not all the same, then an error has been detected. Maximum likelihood decoding boils down to **majority voting**: take the bit value that occurs most often among the three copies received. This way, every single-bit error in a any codeword can be corrected.

  The rate of this code is $1/3$. The bit error probability is reduced quadratically, that is, if the channel's bit error probability is p, then the residual bit error probability is approximately $3p^2$ (see Fig. 4.102).

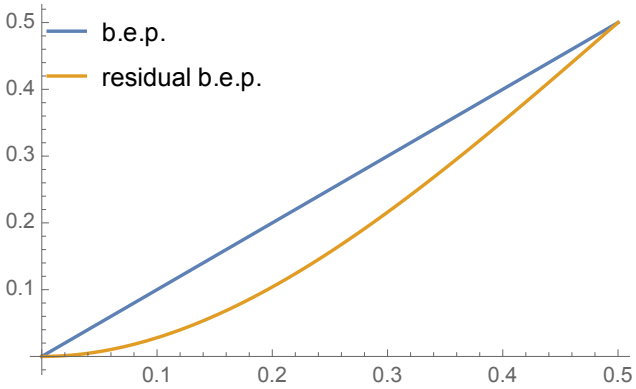

Figure 4.102: Residual bit-error probability (red) compared to bit-error probability (blue) for the repetition code of length 3

  With n repeats, the reliability improves when n increases. However, the efficiency then is $1/n$, and this vanishes when n increases.

- The **Hamming** $(7, 4)$ **code** is a **perfect** error-correcting code that encodes 4 source bits into 7 channel bits, by including 3 so-called parity bits. The sender and receiver act as follows.

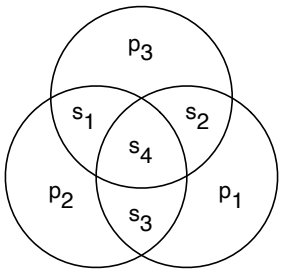

Figure 4.103: Diagram to compute parity bits and error position for the Hamming $(7, 4)$ code

  – The *sender* places the 4 source bits $s_1s_2s_3s_4$ in the diagram of Fig. 4.103, and then determines the 3 **parity bits** $p_1p_2p_3$ such that the number of 1s in each circle is even. This can always be done in exactly one way. The codeword to be sent consists of the sequence[12] $s_1s_2s_3s_4p_1p_2p_3$.

[12]The 7 code bits can be ordered differently, but the receiver must know the order. In this course, we adhere to the indicated order.

– The *receiver* places all received bits in the diagram as indicated, and determines the **parity** of the number of 1s inside each circle, that is, whether that number of 1s is even (represented by 0) or odd (represented by 1). An erroneous bit, if any, is located *outside* all circles with an *even* number of 1s, and *inside* all circles with an *odd* number of 1s. Outside all three circles means there was no error, and otherwise precisely one of the 7 bits is identified as erroneous.

  The combination of the 3 computed parities is known as the **syndrome**. There are 8 possibly syndromes, where all even (000) indicates that there was no error, and each of the 7 other syndromes points at one of the 7 bit positions where the error occurred (under the assumption that at most one bit is in error). Note that 3 bits are exactly enough to identify one of 8 possibilities: no error, or one error bit among 7 transmitted bits. That this works out so nicely is what makes the Hamming $(7, 4)$ code perfect.

Th Hamming $(7, 4)$ code can correct every single-bit error in any codeword. Its rate is $4/7$. For small bit error probabilities p, the residual bit error probability is approximately $6.5p^2$ (see Fig. 4.104).

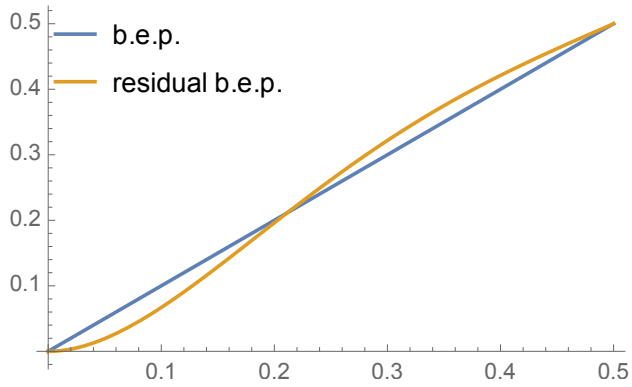

Figure 4.104: Residual bit-error probability (red) compared to bit-error probability (blue) for the Hamming $(7, 4)$ code

N.B. If the Hamming $(7, 4)$ code is used as error-correcting code, on a binary symmetric channel with bit-error probability $p \geq 0.21133$, then the residual error probability will actually exceed p. That is, one is then better off without doing error correction.

Wikipedia Error detection and correction — Forward error correction — Decoding methods — Repetition code — Parity bit — Check digit — Checksum — Hamming $(7, 4)$ — Hamming code

4.11 Bounds on Error Detection and Correction

Definition The **Hamming distance** between two symbol sequences of equal length is the number of positions in the sequences at which corresponding symbols differ.

The **minimum distance** of a code (set of codewords) C is the smallest Hamming distance between any two distinct codewords from C.

Properties

1. Hamming distance is a so-called *metric*, because it satisfies the following properties. Let x, y, and z be symbol sequences of equal length, and let $d(x, y)$ denote the Hamming distance between x and y. **Extra**

   - Hamming distance is non-negative: $d(x, y) \geq 0$.
   - Hamming distance equals zero when two sequences are equal: $d(x, y) = 0$ if and only if $x = y$.
   - Hamming distance is symmetric: $d(x, y) = d(y, x)$.
   - Hamming distance satisfies the **triangle inequality**: $d(x, z) \leq d(x, y) + d(y, z)$.

2. For binary sequences x and y of equal length, the Hamming distance between x and y equals the number of 1s in $x \oplus y$, where addition $\oplus$ is done bitwise modulo 2. Using the weight function w, see (4.94), this equals $w(x \oplus y)$.

3. For a given code C, its ability to detect or correct errors can be quantified as follows.

   Bound on error detection C can be used to *detect* all k-symbol errors in every codeword, if and only if the minimum distance of C is at least $k + 1$.

   Bound on error correction C can be used to *correct* all k-symbol errors in every codeword, if and only if the minimum distance of C is at least $2k + 1$.

   Hence, if the minimum distance of C equals d, then C can be used to

   - *detect* up to $d - 1$ errors per codeword, and
   - *correct* up to $\lfloor (d - 1)/2 \rfloor$ errors per codeword. Here, $\lfloor x \rfloor$ is the value of x rounded *down* to the nearest integer.

4. The minimum distance of some well-known codes:

   - The repetition code that sends k copies of each symbol has minimum distance k.
   - The binary code with one parity bit has minimum distance 2.
   - The Hamming $(7, 4)$ code has minimum distance 3.

5. For the binary symmetric channel, **maximum likelihood decoding** (§4.10) boils down to finding the codeword closest to the received word, in terms of Hamming distance. This is also known as **minimum-distance decoding** and **nearest-neighbor decoding**.

Wikipedia Hamming distance

4.12 Capacity of a Noisy Channel

Definition Consider a binary channel. In the ideal situation (100% reliability), it can transmit 1 bit of information per channel bit sent. According to Shannon's Source Coding Theorem (§4.7), an information source with entropy < 1 can be compressed such that each message can be encoded (on average) in (less than) one bit. This also holds for some (but not all) information sources with entropy $= 1$; for instance, when the source has a two-symbol alphabet with equal probabilities of 0.5.

A noisy channel (§4.9) is not 100% reliable. The noise can be viewed as **anti-information** that reduces the capacity of the channel to transfer information.

The (effective) **channel capacity** of a noisy binary channel is defined as $1 - \mathcal{H}(\text{noise})$, measured in bits, where the noise is a modeled (§4.9) as an information source that produces a 1 to represent the occurrence of an error, and a 0 for the absence of an error.

The relevance of channel capacity becomes clearer in Shannon's Channel Coding Theorem (see §4.13).

Properties

1. The effective capacity of a binary symmetric channel with error probability p is $1 - \mathcal{H}(p)$. See (4.62) in §4.6 for the definition of $\mathcal{H}(p)$, which is the entropy of a binary memoryless source with $P(1) = p$ and $P(0) = 1 - p$.

Wikipedia Channel capacity

4.13 Channel Coding Theorem, Error Reduction

Theorem Shannon's **Channel Coding Theorem** (1948) provides a bound on how much efficiency must be sacrificed to make a noisy channel arbitrarily reliable.

> On average, each bit of a binary information source can be communicated reliably across a noisy channel using an encoding with rate $\approx C$ and not with with a rate $> C$, where C is the capacity of the channel.

More precisely,

> Given a binary information source S and noisy channel with capacity C,
>
> - for every desired code rate $R < C$ and desired bound $\varepsilon > 0$ on the residual error probability, there exists a pair of encoding-decoding algorithms with rate $\geq R$ for sending S over C, such that the residual error probability is $< \varepsilon$, and
> - every family of encoding-decoding algorithms for which the residual error probability approaches 0 has a rate converging to $\leq C$, that is, no encoding-decoding algorithms with rate $> C$ can achieve arbitrarily good reliability.

Notes on Shannon's Channel Coding Theorem:

1. The theorem provides a precise bound that works in *two directions*: Arbitrarily reliable communication at a rate higher than the channel capacity is impossible, and the channel capacity can be approached from below as closely as one wants.
2. The theorem can be combined with the Source Coding Theorem (§4.7) as follows. Given a binary source with entropy $\mathcal{H}$ and binary channel with capacity C, then for every $\varepsilon > 0$ and rate $R < C/H$ there exists an encoding-decoding with rate $\geq R$ such that the residual error probability is $< \varepsilon$. If the channel is reliable ($C = 1$), then this boils down to the Source Coding Theorem. If the source is already fully compressed ($\mathcal{H} = 1$), then this boils down to the Channel Coding Theorem.

3. The theorem does not promise error-free transmission. It only states that the residual error probability can be made as small as desired.

4. Which error correction algorithm will work depends, in general, on the desired level of reliability (the choice of ε). To achieve a higher reliability, that is, a smaller value of ε, requires a more advanced error correction algorithm. This may in turn require that more symbols are combined and encoded together, generally incurring more overhead costs per message.

5. In fact, the proof of the theorem is based on increasing the input block length of the encoding, to encode multiple symbols together. The code words can basically be assigned randomly, using maximum likelihood decoding (§4.10).

6. Achieving a higher reliability (smaller ε) may require an increase in communication **latency**.

7. The theorem motivates the relevance of channel capacity, as defined in §4.12.

8. The practical challenge is to strike a balance between

   - improved communication reliability,
   - complexity and efficiency of the encoding-decoding algorithms, and
   - loss of efficiency in the communication.

   Ideal are simple and fast encoding-decoding algorithms that achieve a high reliability at minimum loss of efficiency.

   Good practical codes do exists and are applied in all modern communication equipment, ranging from smartphones, to cable modems, to satellites, to space probes. However, they are beyond the scope of this course.

Wikipedia Noisy-channel coding theorem

4.14 Secure Communication and Storage: Protection against Adversaries

Definitions **Information security** is concerned with communication and storage in the presence of **attackers** that try to interfere on purpose, in contrast with noise that acts without purpose (§4.8). **Cryptography** is the discipline that studies techniques to accomplish information security. In contrast to efficiency and reliability, it is not so easy to characterize desirable properties for information security, because there are many aspects to consider.

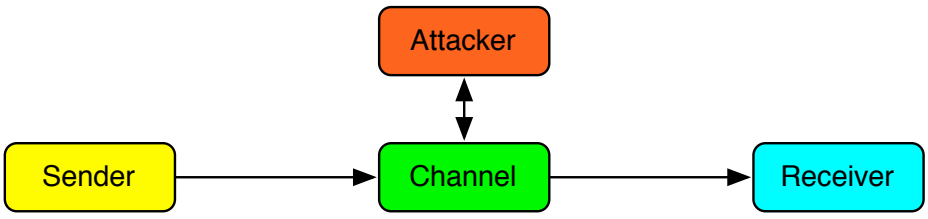

Figure 4.141: A communication channel under attack by an adversary

Information security concerns addressed in this course are:

Confidentiality meaning that the attacker cannot extract any information (also not partially) from messages sent on the channel, that is, messages are only intelligible to the sender and intended receiver. This is also referred to as **secrecy** and **privacy**.

Authenticity meaning that the attacker cannot pretend to be someone else, that is, send a message in the name of someone else, without this being noticed by the receiver.

Integrity meaning that the attacker cannot modify messages on the channel, such as replacing (part of) a message with other content, without this being noticed by the receiver.

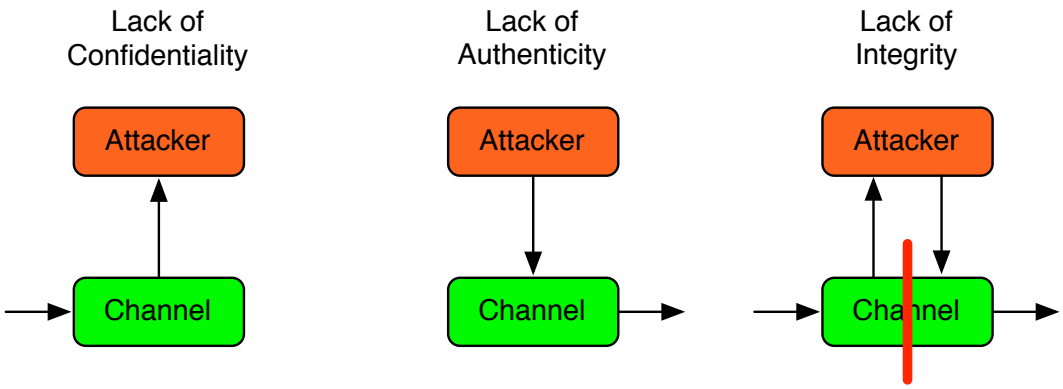

Figure 4.142: Three information security concerns

Many people perceive confidentiality as the most important goal of cryptography, but the importance of authenticity and integrity should not be underestimated. In many cases, these are in fact more important than confidentiality.

Note that there are other concerns as well, such as **availability** (the inability to deny access to resources), **non-repudiation** (the inability to deny that a message was indeed sent), and **anonymity** (the inability to trace messages back to the sender).

Various degrees of security can be distinguished, including:

- **perfectly secure** or **unconditionally secure**: Even if the attacker has unlimited resources, security cannot be broken, not even partially.
- **computationally secure**: The attacker cannot break security, not even partially, with current or future computational resources; for instance, a **brute-force attack** that tries all keys should be practically impossible (though not theoretically impossible), mainly because it would take too much time).
- various degrees of breakable, ranging down to
- completely insecure, where the channel is an open book to the attacker.

In practice, there is the economic perspective:

- Security is economically sufficient when the cost to the attacker of achieving a (partial) compromise of security is considerably higher than the loss of value to the sender and receiver due to the (partial) compromise.

Applications Here are some more advanced applications that require information security:

- secure mobile telephony (e.g. GSM) and internet (wired and wireless)
- contactless electronic car keys, hotel/office door keys, and passports
- ecommerce and internet banking
- copyprotected ebooks, music, and videos
- anonymous digital money
- electronic voting

Extra Terminology: Besides cryptography, you also encounter the terms **cryptanalysis**, which is the study of techniques to break cryptographic systems, and **cryptology**, which refers to the union of cryptography and cryptanalysis.
Khan Academy on YouTube: Gambling with Secrets (Cryptography)

Wikipedia Information security — Cryptography — Cryptanalysis

4.15 Encryption, Decryption, Digital Signature, Cryptographic System

Definitions Also see Fig. 4.151.

Alice Nickname for the generic sender

Bob Nickname for the generic receiver

Eve Nickname for the generic attacker (derives from 'eavesdropper')

Plaintext The original message as chosen by the sender, and intended to arrive at the receiver; also known as **cleartext**

Encryption An encoding for a cryptographic purpose, especially for confidentiality; usually involving some other information, called an encryption **key**; also known as **encipherment**

Ciphertext The result of encrypting a plaintext

Digital signature Information derived by the sender from a message through an encoding (called *signing a message*), in such a way that (a) only the sender can produce that information for this message, and (b) others can easily verify that the information concerns this combination of message and sender; addresses authenticity, integrity, and non-repudiation; usually involves some other information, called a signature **key** (for creating the signature) and a verification key (to verify the signature)

Decryption The inverse of encryption, usually involving some other information (a decryption **key**), possibly different from the key used for encryption; also known as **decipherment**

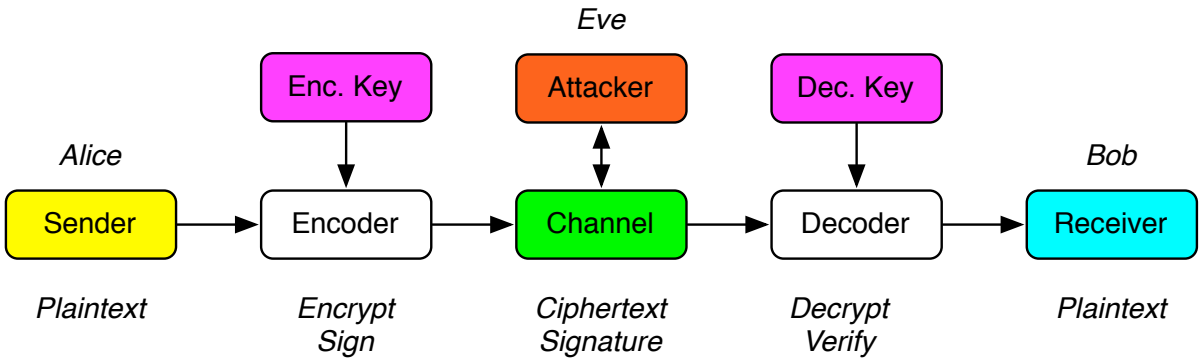

Figure 4.151: Concepts involved in information security

A **cryptographic scheme** (or system) consists of all the facilities that together achieve the desired information security objective. Cryptographic schemes are constructed from various **cryptographic algorithms** that perform input-to-output transformations (like encryption), and **cryptographic protocols** that prescribe how multiple parties should interact to achieve certain information security subgoals (like a key exchange to create a shared secret key over an insecure channel). We use the term **cryptosystem** for a combination of key-generation+encryption+decryption algorithms.

Extra Terminology: A **passive** attacker only taps information from the insecure channel, but does not interfere with algorithms and protocols. An **active** attacker may also interfere with algorithms and protocols, for instance, by deleting, inserting, or modifying messages on the insecure channel.

Wikipedia Alice and Bob — Plaintext — Encryption — Key (cryptography) — Ciphertext — Digital signature — Cryptographic primitive — Cryptographic protocol — Cryptosystem

4.16 Kerckhoffs' Principle

Auguste Kerckhoffs (1883) stated that "a cryptosystem should be secure, even if everything about the system, except the key, is public knowledge". This is known as **Kerckhoffs' Principle**. The security should only be in the difficulty of discovering the secret key, which serves as an extra parameter to the cryptographic algorithms.

Claude Shannon formulated this later as "the enemy knows the system being used", and "one ought to design systems under the assumption that the enemy will immediately gain full familiarity with them" (**Shannon's Maxim**).

When one attempts to achieve security by keeping the cryptographic algorithms secret[13], we speak of **security through obscurity**. In most cases[14], this is a bad idea. Sooner or later, design details of the employed cryptographic system will become known to the enemy. And it usually is very costly to change or replace a cryptographic system. On the other hand, it should be relatively easy to generate new secret keys.

[13] It would be better to write here 'by hoping to keep' them secret.

[14] A recent development in cryptography is that of **obfuscation**. Obfuscation is a provable technique to make the implementation of a (cryptographic) function inaccessible for analysis. The only thing one can do with an obfuscated function is to apply it. This could be called *security by obfuscation*. Note however that current implementations are not practical.

Notes

1. The key that Kerckhoffs mentions in his principle must be selected *randomly*, with a uniform probability distribution; in particular, the key should have a high entropy (§4.6), to avoid giving away information to attackers.

   The generation of random symbols for a key is not easy. In particular, it is important to understand that **statistical randomness** and **cryptographically secure randomness** are not the same. There are simple algorithms to generate **pseudorandom numbers** with good statistical properties. However, once you have observed a few of these numbers, you can easily predict all subsequent numbers. And an attacker can do that as well; so, such a generator is not cryptographically secure (though it is useful for simulations).

2. One might be tempted to use a *compression algorithm* for encryption to guarantee *confidentiality*. The reason one might believe this to work is that the output of a good compression algorithm looks quite random, since it has a high entropy. However, the details of the employed compression will leak, giving attackers the possibility to break the encryption. Looking random, and being a good encryption are two different things. Note, however, that a well encrypted message often looks random. This concerns the same difference mentioned in the preceding note between *cryptographically secure randomness* (as obtained by good encryption) and *statistical randomness* (as obtained by good compression).

3. One might be tempted to use an *error-detecting code* to guarantee *integrity*. The reason one might believe this to work is that (a considerable number of) changes in the output of an error-detecting code are detectable. However, the details of the employed code will leak, giving attackers the possibility to forge signatures. Protection agains random noise and against a intelligent adversary are two different things.

4. Do not try to invent your own cryptosystem. Rather, rely on existing standards that have withstood thorough cryptanalysis. We present some options in the next few sections.

Wikipedia Kerckhoffs's Principle

4.17 One-Time Pad

Definition The so-called **one-time pad** (OTP) is a simple cryptosystem that provides **perfect secrecy** (confidentiality) [2, Ch.8]. The sender and receiver must somehow agree on a fresh random **key** that uses the same alphabet as the plaintext and that has the same length as the plaintext.

OTP encryption of the plaintext is done symbol by symbol. We explain it using bits. If m is the plaintext message bit and k the random key bit, then the ciphertext bit c is obtained as

$$c \;=\; m \oplus k \tag{4.171}$$

where $\oplus$ stands for **addition modulo 2** (that is, using clock arithmetic, ignoring multiples of 2) defined by (4.92). We will also write

$$ciphertext \;=\; plaintext \oplus key \tag{4.172}$$

where $\oplus$ is then applied per symbol. Fig. 4.172 illustrates this with image encryption. For an alphabet of size N, you would use addition modulo N, associating each symbol with a unique value from 0 through $N-1$. OTP decryption works similarly, using *subtraction* modulo N.

Figure 4.171: A 128×128-image encrypted by a random one-time pad key: white is 0, black is 1

Properties of the one-time pad:

1. OTP encryption and decryption are very simple and efficient. In fact, on relatively short messages, they can be carried out manually, without computer support.

2. In the case of bits, OTP decryption is the same as encryption:

$$\begin{aligned}
& \mathit{ciphertext} \oplus \mathit{key} \\
= & \quad \{ \text{ definition of } \mathit{ciphertext} \ \} \\
& (\mathit{plaintext} \oplus \mathit{key}) \oplus \mathit{key} \\
= & \quad \{ \oplus \text{ is associative: order of evaluation is irrelevant } \} \\
& \mathit{plaintext} \oplus (\mathit{key} \oplus \mathit{key}) \\
= & \quad \{ \text{ each bit is its own inverse under } \oplus\text{: } x \oplus x = 2x = 0 \ \} \\
& \mathit{plaintext} \oplus 0 \\
= & \quad \{ \ 0 \text{ is the identity element of } \oplus\text{: } 0 \oplus 0 = 0,\ 1 \oplus 0 = 1 \ \} \\
& \mathit{plaintext}
\end{aligned}$$

3. In "the old days", spies were sent behind enemy lines with a notepad that had sheets full of random numbers. Each sheet was used only once and then destroyed. This explains the name 'one-time pad'.

4. OTP provides perfect secrecy, under the assumptions that the key is random and unknown to the attacker. Assume that the attacker intercepts the ciphertext c. What are the possible plaintexts that it could have originated from? The following argument shows that any text with the same length as the ciphertext (and hence as the original plaintext) could have resulted in the intercepted ciphertext. Let m be any message with the same length as c. When this m is encrypted with the key $m \oplus c$, one obtains:

$$\begin{aligned}
& \mathit{Encrypt}_{OTP}(m, m \oplus c) \\
= & \quad \{ \text{ definition of OTP encryption (4.172) } \} \\
& m \oplus (m \oplus c) \\
= & \quad \{ \oplus \text{ is associative: order of evaluation is irrelevant } \} \\
& (m \oplus m) \oplus c) \\
= & \quad \{ \text{ each bit is its own inverse under } \oplus\text{: } x \oplus x = 2x = 0 \ \} \\
& 0 \oplus c \\
= & \quad \{ \ 0 \text{ is the identity element of } \oplus\text{: } 0 \oplus 0 = 0,\ 0 \oplus 1 = 1 \ \} \\
& c
\end{aligned}$$

Thus, from the attacker's point of view, any message m with the same length as the ciphertext could have been the original plaintext. Even a **brute-force attack** that tries all keys would not help, because many of the possible texts will make semantically good candidates.

5. OTP is, in essence, the only scheme that provides perfect secrecy. In other words, all other schemes are, in theory, breakable.

6. OTP requires that the key is known to both the sender and the receiver (but not the attacker). Hence, the key must at some point (either in advance, or afterwards) be communicated *securely*, posing a kind of *chicken-egg problem*. Note that the key has the same length as the ciphertext; thus, the key communication problem is 'equally big' as the message communication problem.

7. OTP needs a *fresh random* key symbol for *each* plaintext symbol to be encrypted. As observed in Note 1 of §4.16 (Kerckhoffs' Principle), generating such random symbols is costly.

8. Reusing a key reduces OTP's level of security considerably. If (binary) messages m_1 and m_2 have both been encrypted by secret key k, obtaining ciphertexts $c_1 = m_1 \oplus k$ and $c_2 = m_2 \oplus k$, then the attacker can calculate $c_1 \oplus c_2$, which equals $m_1 \oplus m_2$, since

$$c_1 \oplus c_2 = m_1 \oplus k \oplus m_2 \oplus k = m_1 \oplus m_2 \oplus 2k = m_1 \oplus m2 \tag{4.173}$$

This reveals at what positions the two plaintext messages agree. Thus, for each pair of corresponding ciphertext bits, there are now only two instead of four possible plaintext bits combinations. If a bit in $c_1 \oplus c_2$ equals 0, then the corresponding plaintext bits are equal, that is, either both 0 or both 1, and if that bit in $c_1 \oplus c_2$ equals 1, then the corresponding plaintext bits differ. Either way, the number of possibilities halved. When reusing the key one more time, the number of possibilities even drops by a factor four.

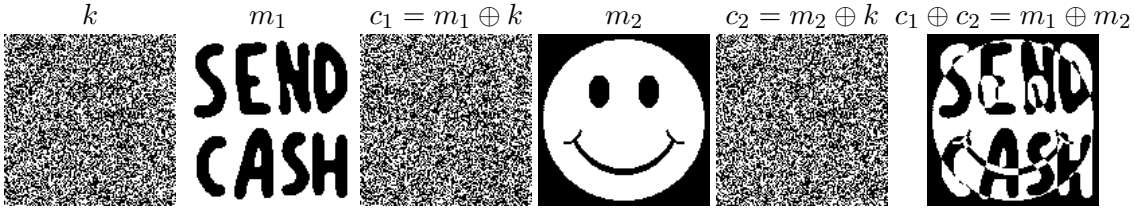

Figure 4.172: Two distinct messages OTP-encrypted by the same key, and their sum

Fig. 4.172 illustrates this with image encryption.[15] A white pixel corresponds to a 0, and a black pixel to a 1. It is clear that $c_1 \oplus c_2$ 'leaks' information about both plaintexts.

9. OTP encryption can be viewed as adding *noise* with bit-error probability $1/2$ and hence entropy 1 (see §4.9, Property 1). The ciphertext resulting from encryption is the channel output, which is basically just noise that conveys no information whatsoever about the channel input, which is the plaintext. This noise is visible in the keys and ciphertexts in Fig. 4.171 and 4.172.

10. In OTP, both the key and the resulting ciphertext are random, implying that neither can be compressed. This property holds for all good encryption algorithms.

[15] `http://cryptosmith.com/2008/05/31/stream-reuse/`

11. OTP encryption can be used for perfect **secret sharing**, where a secret is split into two separate **shares**, such that each individual share conveys no information about the secret, and the two shares together provide enough information to reconstruct the secret. The (random) key and the ciphertext obtained by encrypting the secret with that key form two such shares.

12. OTP offers **deniable encryption**, in the following sense. When a ciphertext is intercepted by an attacker, and the sender or receiver is asked (read: tortured) to reveal the key so as to expose the plaintext, then they can trick the attacker and give him a key (different from the original key) that will decrypt the ciphertext into a carefully chosen innocent text (that differs from the original plaintext). In other words, the sender and receiver can deny access to the original plaintext and instead divert the attacker to something else. The attacker has no way of verifying that the exposed key was not the original key. This is a direct consequence of the argument that proves perfect secrecy (Property 4 above).

Extra There are many variants for secret sharing:

- You can distribute the secret over more than two parties by including an additional random key for each extra party. The ciphertext is obtained by adding all keys to the plaintext:

$$ciphertext \quad = \quad plaintext \oplus key_1 \oplus \cdots \oplus key_{K-1} \tag{4.174}$$

  thus obtaining K **shares**, viz. $key_1, \ldots, key_{K-1}, ciphertext$, that are all needed to reconstruct the secret. This scheme offers perfect secrecy: possessing fewer than K shares provides no information whatsoever about the secret.

- There are schemes for secret sharing among K parties such that, for a given **threshold** T with $1 < T < K$, every subset of T parties can reconstruct the secret, but subsets that are smaller than T cannot do so.

Wikipedia One-time pad — XOR cipher — Cryptographically secure pseudorandom number generator — Secret sharing — Deniable encryption

4.18 Symmetric (Secret-key) Cryptosystems

Definitions In **symmetric cryptosystems**, the sender and receiver use the *same* **key** in their cryptographic algorithms, such as encryption and decryption (see Fig 4.181). This key is a piece of information that must not become known to attackers. It is a **shared secret**, often called **secret key**.

The secret key must be communicated securely, either before or after sending the message, either from sender to receiver, or from receiver to sender. Encryption is done through a function $E_k(m)$ that maps plaintext message m and key k to a ciphertext. Similarly, decryption is done through a function $D_k(c)$ that maps ciphertext c and key k to a plaintext. For deterministic[16] encryption, we need D_k to be the inverse of E_k, in that,

$$E_k(m) = c \iff D_k(c) = m \tag{4.181}$$

[16]For probabilistic encryption, the implication from right-to-left ($\Leftarrow$) does not hold, to avoid that an attacker can recognize that some message was sent twice by comparing intercepted ciphertexts. Also see below under mode of operation. Probabilistic encryption is even more relevant for asymmetric cryptosystems (§4.20).

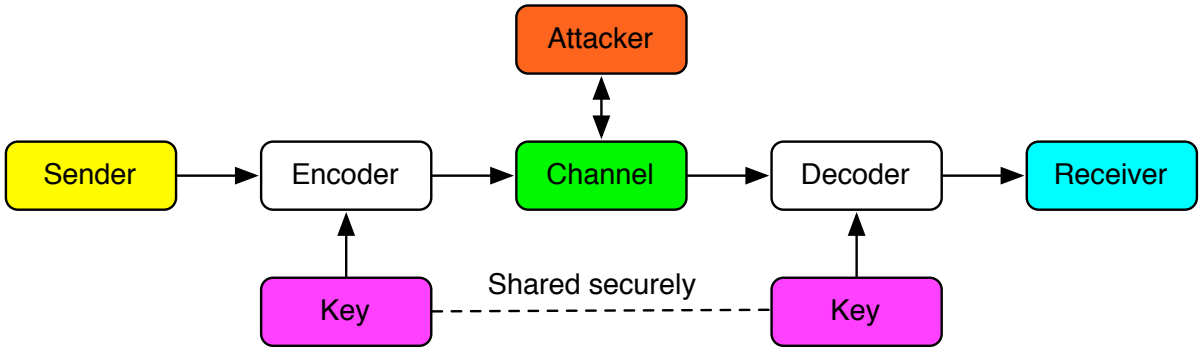

Figure 4.181: Concepts involved in symmetric crypto; there is one secret shared key

The one-time pad (§4.17) is symmetric in the sense that it uses a secret key that is shared between sender and receiver. It should, however, be noted that with the one-time pad every message requires a fresh key that has the same length as the message. To overcome these limitations, symmetric encryption usually involves a fixed-length block code (§4.3), usually referred to as **block cipher**, which is parameterized by a fixed-length key. All blocks are encrypted using the same key, and that same key is used again for other messages.

Classical encryption schemes involve just simple **substitutions**, where each plaintext symbol is replaced by a ciphertext symbol, or **transpositions**, where the plaintext symbols are permuted. Details of these classical schemes are beyond the score of this course. **Extra**

Good encryption algorithms work by **diffusion** and **confusion**, two terms coined by Shannon and explained in his, then confidential, report *A Mathematical Theory of Cryptography* from 1945:

> "Two methods (other than recourse to ideal systems) suggest themselves for frustrating a statistical analysis. These we may call the methods of *diffusion* and *confusion*. In the method of diffusion the statistical structure of M [the plaintext message] which leads to its redundancy is "dissipated" into long range statistics—i.e., into statistical structure involving long combinations of letters in the cryptogram [the ciphertext].
>
> "The method of confusion is to make the relation between the simple statistics of E [the ciphertext] and the simple description of K [the key] a very complex and involved one."

Good confusion means that, for each key, the mapping between plaintexts and ciphertexts behaves as a random permutation, and that these permutations are independent for distinct keys. Good diffusion means that a small change in the plaintext or the key brings about a large and unpredictable change in the resulting ciphertext: every ciphertext bit should depend on all plaintext bits and key bits. More precisely, when a single plaintext bit or key bit changes, every ciphertext bit should change with probability $1/2$ (this is also known as the **avalanche criterion**).

How to achieve diffusion and confusion effectively and efficiently is beyond the scope of this course. Addition modulo 2 of a random key as done in the one-time pad (4.17), is a simple way of creating confusion. Nowadays, symmetric encryption algorithms consist of multiple rounds of substitutions and permutations, combining various weaker algorithms to produce a stronger algorithm. **Extra**

When using block ciphers, it is important to use them well. Here we briefly discuss the following two **modes of operation** (also see [2, Ch.8]).

Electronic Codebook (ECB): Each plaintext block is encrypted separately, with the same key (see Fig. 4.182).

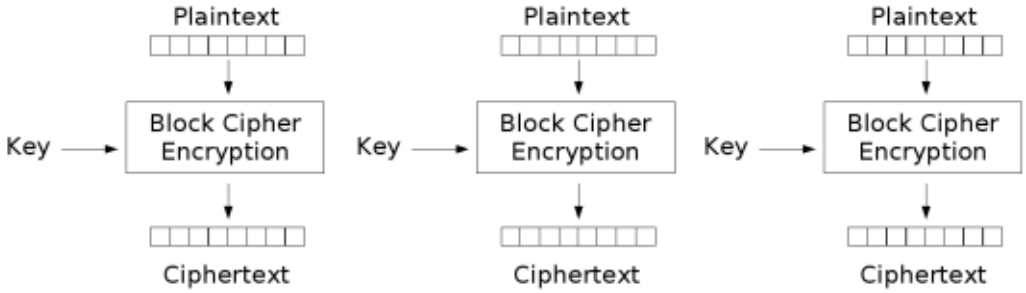

Figure 4.182: Encryption for a block cipher using electronic codebook (ECB) mode of operation[17]

This way, multiple blocks can be encrypted (and also decrypted) in parallel. However, the ECB mode is *not secure*, as illustrated in Fig. 4.185. If two plaintext blocks are identical, then so are their encryptions. The diffusion range is limited. This allows the attacker to gain information from the ciphertext about the plaintext. In particular, if the same message is sent again, it will use the exact same ciphertext (remember that the key is reused). This is also undesirable, since it might be informative to the enemy.

Cipher-Block Chaining (CBC): To overcome the limitations of the straightforward ECB mode, a (fresh random) **initialization vector** is introduced and the encryption of the blocks is no longer done separately (see Fig. 4.183). Before encrypting the first

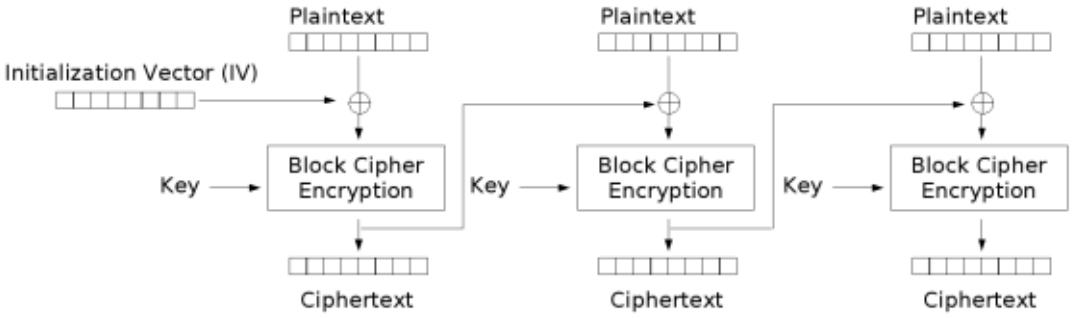

Figure 4.183: Encryption for a block cipher using cipher-block chaining (CBC) mode of operation[17]

plaintext block, it is combined (using $\oplus$, addition modulo 2) with the initialization vector, and before encrypting each subsequent block, it is $\oplus$-combined with the ciphertext of the preceding block. The initialization vector is transmitted together with the ciphertext. Using CBC mode, plaintext blocks cannot be encrypted in parallel, but ciphertext blocks can be decrypted in parallel (see Fig. 4.184).

CBC improves diffusion considerably, since ciphertext blocks now depend on the initialization vector and all preceding plaintext blocks. Fig. 4.185 illustrates this with 128×128-bit images, using a simple cipher that $\oplus$-adds (modulo 2) each 8-bit plaintext block to the 8-bit key and rotates the result over 3 positions to the right.

[17] `https://en.wikipedia.org/wiki/Block_cipher_mode_of_operation`

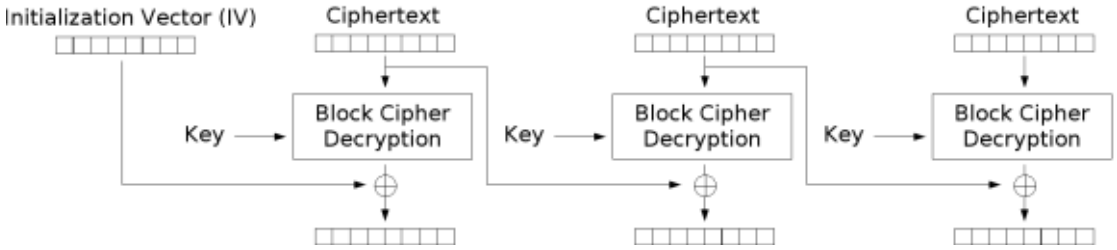

Figure 4.184: Decryption for a block cipher using cipher-block chaining (CBC) mode of operation[17]

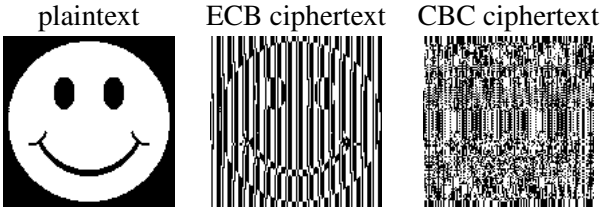

Figure 4.185: Comparison of electronic codebook and cipher-block chaining modes

Properties

1. In contrast to the one-time pad (§4.17), symmetric block ciphers use much shorter keys and therefore do not offer **perfect secrecy**. In particular, given a ciphertext obtained by encrypting a plaintext with a symmetric block cipher, not every possible text of corresponding length can be the original plaintext. Actually, there are considerably fewer possibilities.

2. Also in contrast to the one-time pad (§4.17), symmetric block ciphers do not offer **deniable encryption**. In fact, given a ciphertext obtained by encrypting a plaintext with a good symmetric block cipher, it is quite unlikely that decryption of the ciphertext using any other key than the original key will lead to a sensible plaintext. This is also what makes a brute-force attack possible by trying (all) keys.

For a symmetric block cipher to be secure, we want the following properties.

3. Given a ciphertext $c = E_k(m)$, it must be difficult to find m, without knowing the secret key k. This implies that it must also be difficult to find k when only c is given.

4. Usually, one also wants that given a message m and corresponding ciphertext $c = E_k(m)$, it will be difficult to find the secret key k, because keys are reused. **Extra**

5. In fact, one even wants the much stronger property called **ciphertext indistinguishability**: given the set $M = \{ m_1, m_2 \}$ of two plaintext messages m_1 and m_2 and the set $C = \{ c_1, c_2 \}$ of two corresponding ciphertexts $c_1 = E_{k_1}(m_1)$ and $c_2 = E_{k_2}(m_2)$, it must be difficult to determine how M and C match up, that is, which ciphertext corresponds to which plaintext. Note that the sets M and C are not ordered. This property ensures that ciphertexts do not leak any information about their plaintexts. **Extra**

6. In particular, a **brute-force attack** that tries all possible keys should be practically infeasible. This means that the **key space**, the set of all possible keys, must be sufficiently large. For n-bit keys, the size of the key space is 2^n.

Strengths of symmetric cryptosystems using a block cipher:

7. Is easy to understand and use.
8. Is relatively fast, compared to asymmetric crypto (§4.20).
9. Uses relatively short keys, compared to asymmetric crypto (§4.20).
10. Keys can be reused, in contrast to the one-time pad (§4.17).

Weaknesses of symmetric cryptosystems:

11. Keys must be shared securely between sender and receiver; hence, **key distribution** is a problem. Cf. the chicken-egg problem pointed out for the one-time pad (§4.17, Property 6). Note, however, that in symmetric cryptosystems the key is usually much shorter than the messages and it can be reused. So, here key distribution is a 'smaller' security problem than that of sending the messages.

    When used for secure storage of personal files, the key need not be shared. But it must be stored securely.

12. **Key management** becomes a problem with more users. If a group of N persons wants to keep all pairwise communication confidential from the other $N-2$ persons, then $N(N-1)/2 \approx N^2/2$ shared secret keys are needed. Each person needs to carry out a secure administration of $N-1$ keys and how these keys associate to receivers.
13. It is harder to use symmetric cryptosystems to protect also authenticity and integrity.

Applications There are numerous symmetric cryptosystems, some of which have become standards with widespread adoption.

Data Encryption Standard (DES), an older (1977) standard block cipher, no longer considered secure, and not recommended for new systems. It operates on 64-bit blocks, using a 56-bit key, and involves 16 rounds of substitutions and permutations. A strengthened version, called **Triple DES**, is still applied in electronic payment systems.

Advanced Encryption Standard (AES), a current preferred standard block cipher, in use since 2001. It operates on 128-bit blocks, using a key of 128, 192, or 256 bits, and involves 10, 12, or 14 rounds of substitutions and permutations.

Wikipedia Symmetric-key encryption — Shared secret — Block cipher — Confusion and diffusion — Deterministic encryption — Data Encryption Standard — Advanced Encryption Standard — Block cipher mode of operation

4.19 Three-Pass Protocol

Definition Shamir's **three-pass protocol** is a cryptographic protocol that enables two parties to communicate a message securely using an insecure channel without prior sharing of secret data (such as a key). Fig. 4.191 explains this protocol using one strongbox and two distinct padlocks. Each party has a personal padlock-plus-key, where the key is private and the padlock is public. The main idea behind such a public padlock is that, given a locked padlock,

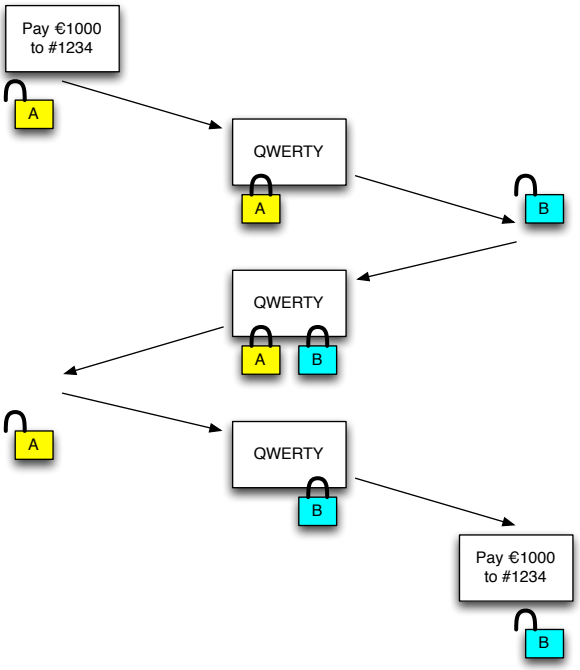

Figure 4.191: The three-pass protocol using two padlocks with private keys

- it is hard to open the padlock without possessing the right key (also, the key cannot easily be reverse-engineered by inspecting the padlock), and
- it is easy to open the padlock with the right key.

Observe that the strongbox is always locked when in transit, ensuring confidentiality of its content.

It is possible to 'digitize' this protocol (also see under Extra). Here is a simple digital version (which turns out to be insecure) to illustrate the main idea. Let m be the plaintext message in binary.

1. (a) Alice generates a private random binary key k_A having the same length as m.
   (b) Alice adds her private key to the message modulo 2, obtaining ciphertext $c_1 = m \oplus k_A$.
   (c) Alice sends c_1 to Bob.
2. (a) Bob generates a private random binary key k_B having the same length as c_1.
   (b) Bob adds (modulo 2) his private key to c_1, obtaining ciphertext $c_2 = c_1 \oplus k_B$.
   (c) Bob sends c_2 back to Alice.
3. (a) Alice adds (modulo 2) her private key to c_2, obtaining ciphertext $c_3 = c_2 \oplus k_A$.
   (b) Alice sends c_3 to Bob.
4. (a) Bob adds (modulo 2) his private key to c_3, obtaining the plaintext m, since

$$\begin{aligned} & c_3 \oplus k_B \\ = & \quad \{ \text{ definition of } c_3 \ \} \\ & c_2 \oplus k_A \oplus k_B \end{aligned}$$

$$
\begin{aligned}
&= \quad \{ \text{ definition of } c_2 \ \} \\
&\quad c_1 \oplus k_B \oplus k_A \oplus k_B \\
&= \quad \{ \text{ definition of } c_1 \ \} \\
&\quad m \oplus k_A \oplus k_B \oplus k_A \oplus k_B \\
&= \quad \{ \oplus \text{ is associative and commutative } \} \\
&\quad m \oplus k_A \oplus k_A \oplus k_B \oplus k_B \\
&= \quad \{ \text{ each bit is its own } \oplus\text{-inverse: } x \oplus x = 0 \ \} \\
&\quad m \oplus 0 \oplus 0 \\
&= \quad \{ \ 0 \text{ is the } \oplus\text{-identity: } x \oplus 0 = x \ \} \\
&\quad m
\end{aligned}
$$

Each ciphertext by itself provides perfect secrecy, as with the one-time pad (§4.17):

$$c_1 = m \oplus k_A \tag{4.191}$$
$$c_2 = m \oplus k_A \oplus k_B \tag{4.192}$$
$$c_3 = m \oplus k_B \tag{4.193}$$

Also, no information about the plaintext can be deduced from any pair of ciphertexts. Unfortunately, an attacker who gets hold of *all three* ciphertexts can compute their $\oplus$-sum, obtaining ...

$$
\begin{aligned}
&\quad c_1 \oplus c_2 \oplus c_3 \\
&= \quad \{ \text{ properties above of } c_i \ \} \\
&\quad m \oplus k_A \oplus m \oplus k_A \oplus k_B \oplus m \oplus k_B \\
&= \quad \{ \oplus \text{ is associative and commutative } \} \\
&\quad m \oplus m \oplus m \oplus k_A \oplus k_A \oplus k_B \oplus k_B \\
&= \quad \{ \ x \oplus x = 0 \text{ and } x + 0 = x \ \} \\
&\quad m
\end{aligned}
$$

... the plaintext. Hence, this version is *not* secure.

Properties of the three-pass protocol

1. The three-pass protocol addresses the chicken-egg problem that was pointed out for the one-time pad cryptosystem (§4.17, Property 6), and it solves the key distribution and key management problems pointed out for symmetric cryptosystems in general (§4.18, Properties 11 and 12). Each sender-message-receiver combination can use a freshly generated (random) secret key shared through the three-pass protocol. Such a key is called a **session key**.

2. The three-pass protocol works because adding and removing padlocks on the strongbox are *commuting* operations: you can add padlock A to an open strongbox, next add padlock B, then remove padlock A, and finally remove padlock B, to obtain an open strongbox again. If the receiver would put the received strongbox-locked-with-the-padlock-of-the-sender *inside* another strongbox and lock that with his padlock, then the sender would not be able to remove her padlock.

Also the (insecure) version based on adding a key modulo 2 works because it involves the *commutative* operator $\oplus$.

3. Because the strongbox is transferred three times, this protocol is not so efficient.

4. Because it is not so efficient, it is best to use it sparingly. For instance, it can be used to transfer a copy of a key for a padlock, which is subsequently used in a symmetric way for multiple exchanges of (possibly large) messages.

5. Its plain form is not secure against *active* attackers. The protocol is vulnerable to the **man-in-the-middle (MiM) attack**, as illustrated in Fig. 4.192. Attackers can trick both sender and receiver by introducing a second, identical looking, strongbox and a third, identical looking, padlock. This way, an attacker not only gets an opportunity to inspect the content of the strongbox (breaking confidentiality), but also to tamper with the content (breaking integrity), for instance to modify it or to replace it altogether.

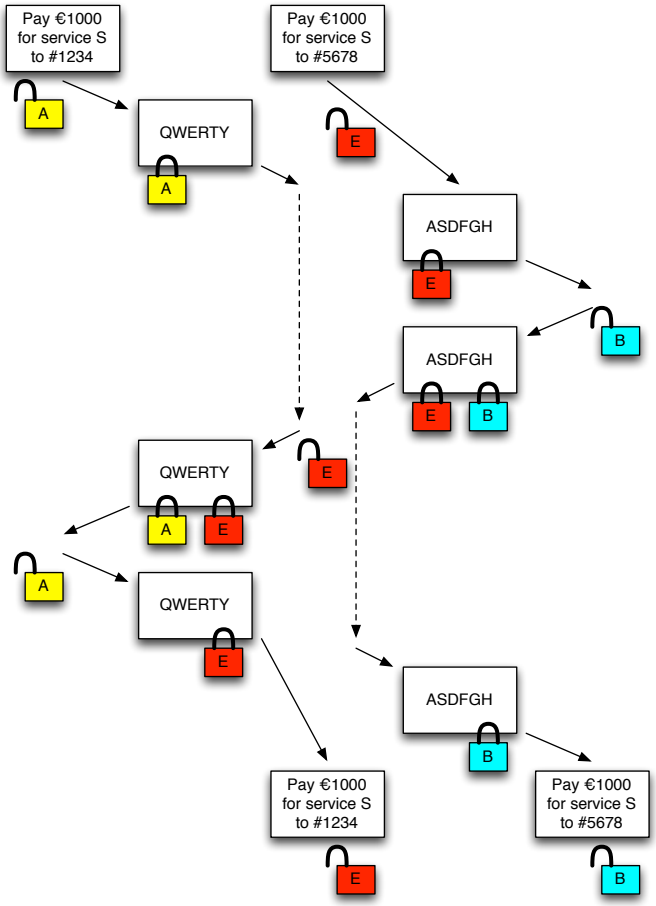

Figure 4.192: Man-in-the-middle attack on the three-pass protocol, using padlocks

Extra

- Shamir presented (around 1980) a digital version of the three-pass protocol that is secure against *passive* attackers (also see Property 5 above). This protocol is also known as Shamir's **no-key protocol**. It is based on **modular exponentiation**[18] of integer values defined by

$$x \;=\; g^a \bmod p \tag{4.194}$$

where $a \bmod b$ for $b > 0$ is the remainder of dividing a by b. Modular exponentiation can be computed efficiently. Alice transmits message m, being a number with $1 < m$, to Bob as follows.

1. (a) Alice generates a sufficiently large random prime p with $m < p$.
   (b) Alice generates a random number a with $1 \leq a < p-1$, such that a and $p-1$ have no common divisors other than 1, that is, $\gcd(a, p-1) = 1$.[19]
   (c) Alice computes $c_1 = m^a \bmod p$.
   (d) Alice sends (p, c_1) to Bob.
2. (a) Bob generates a random number b with $1 \leq b < p-1$, such that b and $p-1$ have no common divisors other than 1, that is, $\gcd(b, p-1) = 1$.
   (b) Bob computes $c_2 = {c_1}^b \bmod p$.
   (c) Bob sends c_2 to Alice.
3. (a) Alice computes a' such that
$$a \cdot a' \;=\; 1 \bmod p-1 \tag{4.195}$$
   which is possible since $\gcd(a, p-1) = 1$.
   (b) Alice computes $c_3 = {c_2}^{a'} \bmod p$.
   (c) Alice sends c_3 to Bob.
4. (a) Bob computes b' such that
$$b \cdot b' = 1 \bmod p-1 \tag{4.196}$$
   which is possible since $\gcd(b, p-1) = 1$.
   (b) Bob computes ${c_3}^{b'} \bmod p = m$, because $1 < m < p$, and calculating modulo p, we have

$$\begin{aligned}
& {c_3}^{b'} \\
= & \quad \{ \text{ definition of } c_3 = {c_2}^{a'} \bmod p \ \} \\
& \left({c_2}^{a'}\right)^{b'} \\
= & \quad \{ \text{ property of exponentiation } \} \\
& {c_2}^{a' \cdot b'} \\
= & \quad \{ \text{ definition of } c_2 = {c_1}^b \bmod p \ \} \\
& \left({c_1}^b\right)^{a' \cdot b'} \\
= & \quad \{ \text{ property of exponentiation } \} \\
& {c_1}^{b \cdot a' \cdot b'}
\end{aligned}$$

[18] Nowadays, mathematicians formulate such protocols in terms of Group Theory, using an algebraic structure called a cyclic group.

[19] There are $\varphi(p-1) \approx p/\log\log p$ candidates to choose from.

$$
\begin{aligned}
&= \quad \{ \text{ definition of } c_1 = m^a \bmod p \ \} \\
&\quad (m^a)^{b \cdot a' \cdot b'} \\
&= \quad \{ \text{ property of exponentiation } \} \\
&\quad m^{a \cdot b \cdot a' \cdot b'} \\
&= \quad \{ \text{ multiplication is commutative } \} \\
&\quad m^{a \cdot a' \cdot b \cdot b'} \\
&= \quad \{ \ a' \text{ and } b' \text{ satisfy (4.195) and (4.196), hence } k \text{ exists } \} \\
&\quad m^{k \cdot (p-1)+1} \\
&= \quad \{ \text{ property of exponentiation } \} \\
&\quad (m^{p-1})^k \cdot m \\
&= \quad \{ \text{ Fermat's Little Theorem: } m^{p-1} = 1 \text{ if } p \text{ is prime and } \gcd(m,p) = 1 \ \} \\
&\quad m
\end{aligned}
$$

The correctness of Shamir's three-pass protocol is, again, based on the *commutativity* of an operator (see Property 2). Its security depends on the assumed[20] difficulty of finding **discrete logarithms**, that is, of solving equation (4.194) for a when x, g, and p are given. Observe, that by similar calculations as above, we find:

$$c_1 = m^a \pmod p \tag{4.197}$$

$$c_2 = m^{ab} = {c_1}^b = {c_3}^a \pmod p \tag{4.198}$$

$$c_3 = m^b \pmod p \tag{4.199}$$

Thus, the ability to find discrete logarithms would let the attacker discover b from c_1 and c_2, and hence b' and m.

- Diffie and Hellman were the first (1976) to publish a practical protocol for establishing a **shared secret** using an insecure channel. The **Diffie–Hellman (DH) key exchange protocol** is also based on **modular exponentiation** (4.194), and works as follows.

  1. (a) Alice takes[21] a large prime p and integer g, such that $1 < g < p - 1$.
     (b) Alice generates a private random number a, being an integer with $1 \leq a < p - 1$.
     (c) Alice computes $x = g^a \bmod p$.
     (d) Alice sends (g, p, x) to Bob.
  2. (a) Bob generates a private random number b, being an integer with $1 \leq b < p - 1$.
     (b) Bob computes $y = g^b \bmod p$.
     (c) Bob sends y back to Alice.
  3. (a) Alice computes $s = y^a \bmod p$.
     (b) Bob computes $x^b \bmod p = s$, because

$$
\begin{aligned}
&\quad x^b \bmod p \\
&= \quad \{ \text{ definition of } x \ \} \\
&\quad (g^a)^b \bmod p
\end{aligned}
$$

[20] We do not know for sure that it is difficult, but currently we cannot do much better than a brute-force search.
[21] To ensure a sufficient security level, the integers g and p must be chosen with some care.

$$
\begin{array}{ll}
= & \{ \text{ property of exponentiation } \} \\
& g^{ab} \bmod p \\
= & \{ \text{ multiplication is commutative } \} \\
& g^{ba} \bmod p \\
= & \{ \text{ property of exponentiation } \} \\
& (g^b)^a \bmod p \\
= & \{ \text{ definition of } c_2 \} \\
& y^a \bmod p \\
= & \{ \text{ definition of } s \} \\
& s
\end{array}
$$

4. Alice and Bob use the commonly agreed number s as **session key** for a symmetric cryptosystem.

Note that the DH protocol involves only *two* transmissions, viz. of (g, p, x) and of y. In this protocol, Alice does not send some particular message to Bob, but rather Alice and Bob agree on a shared secret. Furthermore, they have no direct control over the outcome s; it is 'random', and at least sufficiently random to be cryptographically secure. Note that Alice does not get to know Bob's random number b, and Bob will not know Alice's random number a.

The security of the DH protocol also depends on the assumed difficulty of finding **discrete logarithms**. Actually, it involves a stronger assumption, because only the special case of finding g^{ab} given g^a and g^b needs to be solved.

The function $f_p(g, a) = g^a \bmod p$ serves as a **one-way function** that can be applied efficiently, but not inverted efficiently when only g and p are given. The correctness of the protocol is based on a special *commutativity* property of this one-way function:

$$f_p(f_p(g, a), b) \quad = \quad f_p(f_p(g, b), a) \tag{4.1910}$$

and that for known and fixed g and p, this common value is uniformly distributed, when a and b have a uniform distribution.

Wikipedia Three-pass protocol — Man-in-the-middle attack — Diffie–Hellman key exchange — Modular exponentiation — One-way function

4.20 Asymmetric (Public-key) Cryptosystems

Definitions In **asymmetric cryptosystems**, the sender and receiver use two *different*, but specially related, keys (see Fig. 4.201). The encoding key is the **public key**, and the decoding key is the **private key**.[22] The public key can be published for everyone to see, whereas the private key needs to be kept secret. In fact, the private key need not even be known to both the sender and receiver.

Encryption is done through a function $E_{puk}(m)$ that maps plaintext message m and *public* key puk to a ciphertext. Thus, anyone can encrypt a message for that receiver. Similarly, *decryption* is done through a function $D_{prk}(c)$ that maps ciphertext c and *private* key prk to a plaintext. Thus, only the

[22] For digital signatures, it is the other way round;; see §4.21.

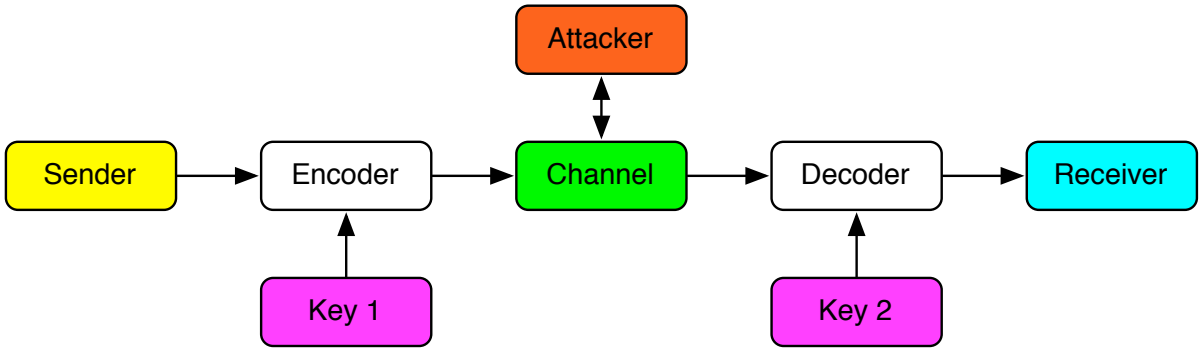

Figure 4.201: Concepts involved in asymmetric crypto; one key is private, the other public

receiver with the private key can decrypt the message. For encryption using a related public-private key pair (puk, prk), we need D_{prk} to be the inverse of E_{puk}, in that,

$$E_{puk}(m) = c \implies D_{prk}(c) = m \tag{4.201}$$

In a way, E_{puk} is a **trapdoor one-way function**, like an open padlock that can be snapped shut without the key in it. You can apply it (lock it), but not undo that (unlock it), without the private key (to activate the trapdoor). This is unlike symmetric encryption, where you need to know the secret key to encrypt, but then you can also decrypt. It is somewhat amazing that such one-way functions with a trapdoor exist.

Such a function is used as follows.

- Each party generates a personal key pair, consisting of a private key and a matching public key. The public key is published for others to use; the private key is kept secret.
- To ensure *confidentiality*, the sender *encrypts* using E_{puk}, where puk is the public key of the receiver. The receiver is then the only one who can decrypt it, being the only one in possession of the matching private key.

Note that even the sender cannot decrypt a message encrypted for someone else.

Properties For asymmetric cryptosystems to be secure, we need the following properties.

1. Given a public key, it must be difficult to find the matching private key.
2. Given a public key puk and a ciphertext $c = E_{puk}(m)$, it must be difficult to find m, without knowing the matching private key prk.
3. Given a public key puk and a message m, it must be difficult to find $D_{prk}(m)$, without knowing the matching private key prk.

For practical asymmetric encryption algorithms (see below), security depends on the (assumed) difficulty of solving certain mathematical problems, such as **integer factorization**, **modular root extraction**, and finding **discrete logarithms**. It is not known whether there exist efficient solutions to these problems, but currently, we have no way of solving these problems in a way that is considerably more efficient that brute force approaches.

Strengths of asymmetric cryptosystems:

4. Private keys are not shared (and kept secret); public keys are published to the world. Thus, **key distribution** is considerably easier than for symmetric cryptosystems.

   When used for secure storage of personal files, the public key need not be published. However, this use is not recommended, since asymmetric cryptosystems are slow (see below).

5. **Key management** is relatively easy. If a group of N persons wants to keep all pairwise communication confidential from the other $N-2$ persons, then N private-public key pairs are needed. All person need to manage, securely, their own private key. They can either request public keys, or store them locally, but this does not have to be done securely.

6. Can also easily protect authenticity and integrity, by means of digital signatures.

Weaknesses of asymmetric cryptosystems:

7. Is conceptually less straightforward, involving different keys that should be used properly.

8. Trustworthiness of public keys is a concern. One important danger with asymmetric cryptography is the **man-in-the-middle (MiM) attack** (also see §4.19, Property 5). When using a public key to encrypt a message or verify a signature, it is important to ensure that the public key indeed belongs to the person you think it belongs to. When you request a public key, the MiM might substitute his own public key, and use his own private key to decrypt (break confidentiality) or create a new message with accompanying digital signature (to break authenticity and integrity). Note that when the MiM possesses the correct public keys of receiver and sender, he can trick both sender and receiver, by re-encrypting and re-signing messages. Also see Fig. 4.192.

   A **Public-Key Infrastructure** (PKI) can address this, by offering authentication for public keys.

9. It requires **probabilistic encryption**, where multiple encryptions of the same plaintext lead to different ciphertexts. **Deterministic encryption**, where (4.201) also holds for the implication from right-to-left ($\Leftarrow$), would allow an attacker to guess plaintexts, encrypt them with the public key, and compare the resulting ciphertexts with the intercepted ciphertext. This way, the attacker can possibly obtain some information without recovering the private key, which is for instance effective when it is known that the message is one of only a few possibilities.

10. Is usually slow; considerably slower than symmetric cryptosystems.

11. Because it is slower, it is usually combined with symmetric encryption; see§4.22.

12. When a private key is exposed, all items encrypted to that key become accessible; typically, all senders use the same public key of the receiver.

13. When the receiver loses the private key, the receiver loses access to all items encrypted to that key; typically, all senders use the same public key of the receiver.

14. Involves longer keys than symmetric cryptosystems to achieve the same level of security.

Applications Some practical asymmetric cryptosystems are the following.

RSA (Rivest–Shamir–Adleman) is an asymmetric cryptosystem involving large prime numbers and modular exponentiation. See [2, Ch.8]. In brief, this is how plain RSA works. **Extra**

Key generation works as follows.

1. Generate two distinct large random primes P and Q. (This is not too difficult.)
2. Compute their product $N = P \cdot Q$. (Easy)
3. Take e coprime to $R = (P-1) \cdot (Q-1)$, that is, e and R have 1 as only common divisor.
4. Determine d such that $e \cdot d = 1 \bmod R$. (Again, easy)
5. Publish (N, e) as public key; keep d as private key. (P and Q can be destroyed, but may be helpful in improving efficiency of decryption.)

Encrypt plaintext m as

$$c \;=\; m^e \bmod N \tag{4.202}$$

Decrypt ciphertext c as

$$m \;=\; c^d \bmod N \tag{4.203}$$

Correctness is based on the mathematical identity $(m^e)^d = m^{ed} = m \bmod N$, using $\varphi(N) = R$ and Euler's Theorem that $m^{\varphi(N)} \bmod N = 1$.

Security depends on the assumed difficulty of **factorizing** N and **extracting modular roots**, that is, solving equation (4.202) for m when c, e, and N are given. Note that in this form, RSA encryption is deterministic, and hence not considered secure.

ElGamal is an asymmetric cryptosystem based on the Diffie–Hellman key exchange protocol (§4.19, Extra). **Extra**

Key generation See Step 1 of DH: Generate a large prime p, an integer g, such that $1 < g < p-1$,and a random number a, being an integer with $1 \le a < p-1$. Computes $x = g^a \bmod p$. Publish (g, p, x) as public key, and keep a as private key.

Encrypt message m with $1 \le m < p$, by generating a random b, computing $y = g^b$ and $s = x^b \bmod p$, and then sends the ciphertext (y, c) where $c = (m \cdot s) \bmod p$ to Alice. Here, s is known as an **ephemeral key**, because it exists only for this particular message.

Decrypt ciphertext (y, c) by first computing the ephemeral key via $s = y^a \bmod p$ and its multiplicative inverse s^{-1} modulo p, and then $c \cdot s^{-1} \bmod p = (m \cdot s) \cdot s^{-1} \bmod p = m$.

Correctness is based on modular arithmetic; in particular, a commutation property of exponentiation.

Security depends on the assumed difficulty of finding **discrete logarithms**, that is, solving equation (4.194). Note that ElGamal encryption is by definition probabilistic (also see Property 9 above).

Wikipedia Public-key encryption — Probabilistic encryption — Man-in-the-middle attack — RSA (cryptosystem) — ElGamal encryption

4.21 Digital Signatures

Definitions In asymmetric digital signature schemes, the sender and receiver use two *different*, but specially related, keys (see Fig. 4.201). The encoding key is the **private key**, and the decoding key

is the **public key**.[23] *Signing* is done through a function $S_{prk}(m)$ that maps plaintext message m and *private* key prk to a pair containing the message and a signature s. Thus, only the owner of the private key can produce this particular signature. *Verification* is done through a function $V_{puk}(m, s)$ that maps a message with signature to a boolean verdict (valid, or not). Thus, anyone with the public key can verify the signature. For digital signatures, we need the relation

$$S_{prk}(m) = (m, s) \implies V_{puk}(m, s) \tag{4.211}$$

where s is the digital signature obtained for message m by signing it with the private signature key, such that the signature can be verified with the public verification key.

To ensure *authenticity* and *integrity*, the sender applies S_{prk} to the plaintext m to obtain a **digital signature** $(m, s) = S_{prk}(m)$, where prk is the sender's private key. Anyone with the sender's public key puk can check this signature by verifying that $V_{puk}(m, s)$ holds. The sender is the only one who could have produced this (m, s). The value s by itself is referred to as a **detached signature**.

Notes

1. Digital signatures do not prevent tampering with messages, but they do make it highly likely that such tampering will be detected. Because a digital signature is limited in size, there are many more messages than there are signatures. Hence, there must be distinct messages having identical signatures. For a good signature scheme, it is unlikely that more than one of these messages makes 'sense'. Thus, it is highly unlikely that on can change the contents of a message such that (a) it is still meaningful (e.g., with a changed amount or bank account number), and (b) it will have the same signature as before.

2. One might be tempted to use an error-detecting code to provide message integrity, just as one might be tempted to use a compression algorithm to guarantee message confidentiality (see §4.16, Note 2). The reason one might believe this to work is that an error-detecting code will detect certain changes. The reason that it does not work, is that error-detecting codes aim at detecting small random changes (as caused by noise). An intentional change could be large. Furthermore, if the error-detecting code is known, then an attacker could change the message and re-apply the error-detecting code to produce a message that looks valid to the receiver.

Applications

1. Plain RSA can readily be used for digital signatures by applying its decryption function to create a signature, and applying its (deterministic) encryption function to verify a signature:

$$S_{prk}(m) = (m, D_{prk}(m)) \tag{4.212}$$
$$V_{puk}(m, s) = (E_{puk}(m) = s) \tag{4.213}$$

Extra

[23] For encryption, it is the other way round; see §4.20.

1. Because public-key digital signature schemes are relatively slow, they are preferably not applied to (large) messages. In practice, digital signatures work by first applying a **cryptographic hash function** to obtain a, relatively short, **hash value** for the message, and then signing that hash value. Because the hash values are relatively short, such a hash function cannot be *injective*; that is, there will exists multiple messages that all have the same hash value. However, such a cryptographic hash function f should be **collision resistant**, that is, given a desired hash value $h = f(m_1)$, it must be very difficult to create another message m_2 such that $f(m_2) = h$, and even more so if m_2 should have some partially chosen content.

2. Two parties that trust each other, can use a symmetric block cipher to produce a digital signature in the following way. First, they agree on a shared secret key (different from the key they use for encryption). A (detached) signature is obtained by applying the block cipher to the message in cipher-block chaining (CBC) mode (§4.18), and then taking the last output block as signature. This is typically called a **message authentication code** (MAC). By the nature of CBC mode encryption, the MAC depends on all message blocks, and is hard to produce without knowledge of the secret key. The disadvantage of this approach is that it works only between the two parties that share the secret key.

3. There is a trade-off concerning the order of encrypting and signing messages.

   - First encrypting the message and *then* signing the resulting ciphertext makes it possible to verify signatures without having the ability to decrypt (and read) the message. Note that it may be necessary to verify signatures along the way multiple times. Having to decrypt the message in order to verify its authenticity and integrity would make this more costly and would pose a security risk.
   - First signing the message and *then* encrypting both the plaintext and the signature makes it possible to keep the identity of the sender hidden from all but the intended receiver (who can decrypt the message, and then see and verify the signature).
   - Finally, one can encrypt the plaintext into a ciphertext *and* separately produce a signature for the plaintext.

   All these orders are applied in practice. For some details, see Wikipedia on **authenticated encryption**.

Wikipedia Digital signature — Cryptographic hash function — Collision resistance — Message authentication code — Authenticated encryption

4.22 Hybrid Cryptosystem

Definition Both symmetric and asymmetric cryptosystems have their own strengths and weaknesses. A **hybrid cryptosystem** combines symmetric and asymmetric cryptosystems, to get the strengths of both and to mitigate their weaknesses. Typically, in a hybrid cryptosystem, a (slow) asymmetric cryptosystem is used to create and communicate

- (relatively short) shared secret session keys for a (fast) symmetric cryptosystem, and
- digital signatures for keys and messages.

The messages are then encrypted/decrypted using the (fast) symmetric cryptosystem. When used in this way, the keys for the symmetric cryptosystem are generated anew for every message. They need not be managed, because these keys are encrypted, using the asymmetric cryptosystem, and discarded after decryption of the message.

Applications

Pretty Good Privacy (PGP) An open standard for secure communication using a hybrid cryptosystem

GNU Privacy Guard An open-source, multiplatform implementation of PGP that integrates with various other applications, such as email clients. `www.gnupg.org`

Wikipedia Hybrid cryptosystem — Pretty Good Privacy — GNU Privacy Guard

4.23 Combining Compression, Noise Protection, and Encryption

Fig. 4.231 shows how to combine all techniques in order to provide secure, reliable, and efficient communication, or storage. Note that the various encodings and decodings must be ordered carefully. Any other order will be inferior. This order can be motivated as follows (we consider the binary case).

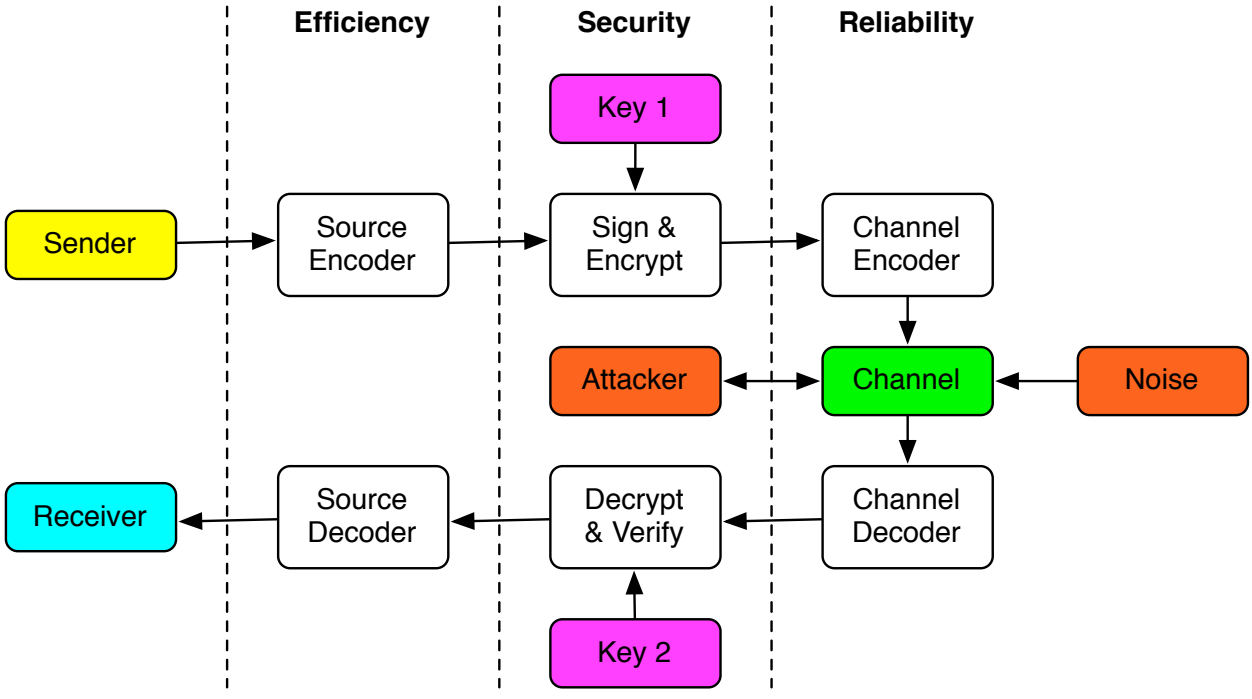

Figure 4.231: Concepts involved in the digital communication stack

- Decoding must always be done in the reverse order of encoding.

- The sender together with the source coder serves as an 'ideal' information source that is very efficient (in its use of bits), having an entropy close to 1. The source encoder removes (almost) all redundancy, and therefore must be 'matched' to the characteristics of the source.
- Compressing before encrypting is beneficial, because compression removes statistical features.
- The channel together the channel encoder and decoder serve as an 'ideal' channel that is still efficient and very reliable. Doing anything to the information after channel encoding and before putting it on the channel would destroy the error control features in the channel code. The channel encoder must be 'matched' to the characteristics of the channel.

In everyday life, this digital communication stack is completely invisible to the general public. Nevertheless it is based on advanced mathematical and computational techniques, that far surpass the theory presented in this course. Here, we have only touched upon the underlying fundamental ideas.

Wikipedia OSI model — 5G

4.24 Modulation and Demodulation

Extra

To communicate and store (abstract) information using physical devices, one needs to cross the boundary between mathematics and physics, between the discrete digital world and the continuous physical world[24].

A **modulator** is a device that converts an (abstract) symbol sequence (such as a sequence of bits) into a physical signal (such as an electromagnetic wave). The modulator converts the symbol sequence to a signal that is suitable for a specific physical medium of communication or storage. This is known as a **line code**.

A **demodulator** is a device that converts a physical signal into a symbol sequence. It does the inverse operation of the corresponding modulator.

A **modem** (short for modulator-demodulator) is a combination of a modulator and a corresponding demodulator, typically used in pairs for two-way communication across some physical medium.

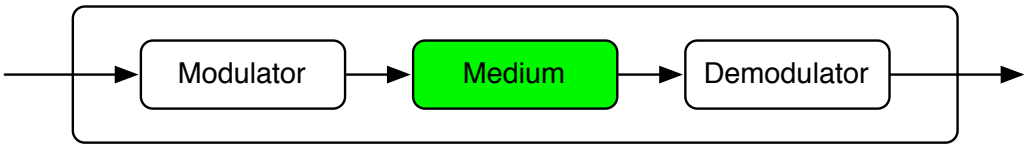

Figure 4.241: Digital channel created from analog medium with modulator-demodulator (modem)

In this context, the **Nyquist–Shannon Sampling Theorem** plays a fundamental role:

> If a continuous signal has no frequency components higher than a given frequency f, then the signal can be completely reconstructed from a sequence of *discrete* samples taken at a frequency of $2f$ or higher.

Wikipedia Modem — Nyquist–Shannon sampling theorem — Line code

[24] Quantum computing may be an exception, in that quantum states are inherently discrete.

5 Limits of Computation

Finite automata (§2.4) are computationally less powerful (§2.6) than Turing machines (§2.9), and according to the Church–Turing Thesis (§2.10) Turing machines are believed to be the most powerful computational mechanisms. However, it turns out that there are well-defined computational problems that are not solvable by any Turing machine. By the Church–Turing Thesis, such problems are then not computationally solvable at all.

Furthermore, even if a computational problem is solvable by a Turing machine (or, equivalently, an algorithm), then it can still be the case that this computational solution is not usable in practice. It turns out that there is a sharp boundary between a class of problems for which we have efficient solutions, and a class of problems for which we (currently) only have inefficient solutions.

See [2, Ch.10]; also see [4, 6].

Wikipedia Limits of computation — Computability theory

5.1 Decidable Decision Problems

Definitions A decision problem (§2.2) is called **decidable** when there exists a Turing machine (§2.9) (or equivalently, an algorithm) that solves it. That is, every instance of the decision problem can be effectively described as an input tape for that Turing machine (algorithm), and the execution of that Turing machine (algorithm) then terminates with the correct yes/no answer.

If a decision problem is not decidable, then we call it **undecidable**.

Properties

- Many syntactic properties of Turing machines and programs are decidable. A syntactic property concerns the description of the Turing machine or program, without executing it. For instance, does a given Turing machine have more than ten states?

Extra Terminology variants: (algorithmically or effectively) solvable

Wikipedia Undecidable problem

5.2 Halting Problem

Definition The **halting problem** is the decision problem, where an instance consists of the description of a Turing machine T and an input tape I for that Turing machine, and the question is whether execution of T on tape I will eventually halt.

Properties

- The halting problem is undecidable (§5.1), that is, there exists no algorithm that solves the halting problem.

- The proof that the halting problem is undecidable roughly proceeds as follows.

  Assume that there exists a Turing machine, say H, that solves the halting problem. We shall show that this leads to a contradiction. What we know about H is that if $\widehat{T}$ is the description of Turing machine T and I is an input tape, then the output of executing $H(\widehat{T}, I)$ answers the question whether execution of T on input tape I halts.

  Using H, we construct a new Turing machine X with the following behavior.

  1. X duplicates the input I on the tape, such that it represents an input for H consisting of I as description of a Turing machine and I as an input for that Turing machine.
  2. X runs H on this tape.
  3. If H reports that the Turing machine described by I halts on input I, then X goes into an infinite loop (never halts), and otherwise X writes 0 on the tape and halts.

  Here is a functional definition of X:

  $$X(I) \quad = \quad \text{if } H(I, I) \text{ then } X(I) \text{ else } 0$$

  Let $\widehat{X}$ be the description of X. What can we infer about the execution of $X(\widehat{X})$? We analyze the two possible cases.

  - The execution terminates. Then this must be because $H(\widehat{X}, \widehat{X})$ returned false, because that is the only way in which X can have terminated. But if H indeed solves the halting problem, then $H(\widehat{X}, \widehat{X}) = \mathit{false}$ means that $X(\widehat{X})$ does not halt. This contradicts the condition for this case.
  - The execution does not terminate. Then this must be because $H(\widehat{X}, \widehat{X})$ returned true, because that is the only way in which X can have avoided termination. But if H indeed solves the halting problem, then $H(\widehat{X}, \widehat{X}) = \mathit{true}$ means that $X(\widehat{X})$ actually halts. This contradicts the condition for this case.

  In either case, we ran into a contradiction. Thus, the assumption that the halting problem is decidable is not tenable.

  The technique used in this proof is called **diagonalization** or a **diagonal argument**, because the hypothetical halting algorithm H, which takes two inputs, is invoked with two identical inputs, and X is given the opposite behavior of H.

  Other famous proofs that rely on diagonalization is Cantor's proof that a set is strictly smaller than its power set (with as special case the proof that the set of natural numbers is strictly smaller than that of the real numbers), and Gödel's proof of his Incompleteness Theorem.

Extra All non-trivial semantic properties of Turing machines and programs are undecidable. This is known as **Rice's Theorem**. Semantic properties concern execution, in contrast to syntactic properties. A property is trivial if it holds for everything or for nothing. Non-trivial here implies that there exists at least one Turing machine that has the semantic property, and another one that does not have that property. Whether a Turing machine halts on a given input is a non-trivial semantic property. Thus, the undecidability of the halting problem is a special case of Rice's Theorem.

Wikipedia Halting problem — Rice's theorem

5.3 Effective Reduction

Definition An **(effective) reduction** (§2.5) of problem P to problem Q is an algorithm that transforms (reduces) each instance of P to a finite number of instances of Q and combines the solutions for the Q-instances into a solution for the P-instance. The existence of such a reduction shows that problem P is no harder than problem Q: if you can solve Q, then you also have a solution to P.

Properties

- A common type of reduction (of problem P to problem Q) is that where a P-instance is reduced to a single Q-instance, whose solution also solves the P-instance. That is, the transformation of the solution of the Q-instance to a solution of the P-instance is the identity transformation.
- The knowledge that a decision problem P is decidable (§5.1) can be used to prove that another decision problem Q is decidable as well, by giving an (effective) reduction of problem Q to problem P: combining the reduction of Q to P with an assumed algorithm for deciding P, we obtain a decision algorithm for Q.
- The knowledge that a decision problem P is undecidable can be used to prove that another decision problem Q is undecidable as well, by exhibiting a reduction of problem P to problem Q. Indeed, if Q would be decidable, then we could combine the reduction of P to Q with an assumed decision algorithm for Q, and thereby obtain a decision algorithm P. But since P is assumed undecidable, it must also be the case that Q is undecidable.
- Typically, we take for P the halting problem (§5.2), that is, we prove undecidability of decision problem Q by reducing the halting problem to Q. Therefore, the halting problem is called the *mother problem* of such undecidable problems.

Wikipedia Reduction (complexity)

5.4 Domino Problem

Definitions A **Wang domino** is a square tile whose edges are labeled. Typically, the labeling is done by colors (see Figure 5.41).

The **domino problem** is the decision problem where an instance consists of a finite set of Wang dominoes, and the question is whether the infinite plane can be tiled by copies of dominoes in the set in such a way that dominoes that are adjacent have the same labels on their shared edge. Note that tiling means that dominoes do not overlap or leave gaps, and adjacent dominoes completely share an edge. Each domino can be used repeatedly (there is an infinite supply), but they may not be rotated or reflected.

Properties

- The domino problem is undecidable (§5.1). This can be proved by reducing (§5.3) the halting problem (§5.2) to the domino problem. This reduction transforms a Turing machine with an initialized input tape into a set of Wang dominoes such that there exists a domino tiling of the (infinite) plane using only dominoes from that set if and only if the Turing machine does not halt.

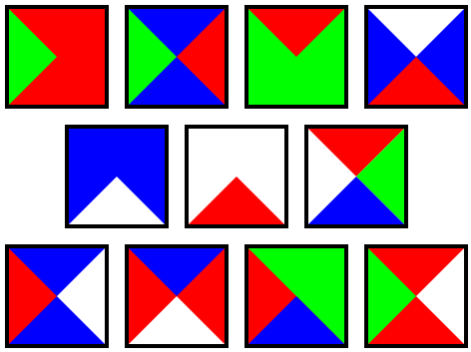

Figure 5.41: Set of 11 Wang dominoes (source: Wikipedia)

Wikipedia Wang tile

5.5 Classes P and NP of Decision Problems

Definitions A decision problem (§2.2) is said to have **polynomial time complexity**, or to be in **class P**, when there exists an algorithm that solves each instance of the decision problem in polynomial time.

A decision problem is said to have **non-deterministic polynomial time complexity**, or to be in **class NP**, when there exists a suitable notion of **certificate** and a polynomial-time algorithm that verifies whether a given certificate proves a yes answer for a given instance.

A problem is said to be **intractable** when it is algorithmically solvable, but the algorithm uses too many resources (time, memory) to be usable in practice. The opposite of intractable is **tractable**.

Properties

1. Class P is a subset of class NP.
2. Decisions problems in class NP typically can be solved by an algorithm that checks all potential certificates for the given instance. The number of potential certificates is finite, but can grow exponentially with the size of the instance. Thus, such an algorithm has an exponential running time, thereby making it unusable in practice.
3. It is not known whether class P equals class NP. This is the famous **P=NP problem**. However, current opinion tends to P $\neq$ NP.
4. Tractable problems are often equated to the class P. Whether problems in class NP are really intractable is not known (P=NP is an open problem). But the best way we know how to solve NP problems makes them currently intractable.

Wikipedia Computational complexity theory — P (complexity) — NP (complexity) — Certificate (complexity) — P versus NP problem

5.6 Traveling Salesman Problem

Definitions There are two common variants of the **traveling salesman problem**, also abbreviated as **TSP**.

- An instance of TSP as an *optimization problem* consists of a set of cities and their mutual distances, and the question is to find a cyclic tour visiting each city exactly once such that this tour is shortest among all possible tours.
- An instance of TSP as a *decision problem* consists of a set of cities, their mutual distances, and a bound k, and the questions is whether there exists a cyclic tour of length at most k visiting each city exactly once.

Properties

- The traveling salesman decision problem is in class NP. A certificate for a yes-instance is a tour, that is, a sequence of cities. It can be checked in polynomial time whether the length of this tour is at most the given bound.

Wikipedia Travelling salesman problem

5.7 Subset sum problem

Definition The **subset sum problem** is a decision problem, where an instance consists of a (finite) set $S$of positive integers and a target number t, and the question is whether S has a subset whose elements sum to exactly t.

Properties

- The subset sum problem is in class NP. A certificate for a yes-instance is a subset of the given set. It can be checked in polynomial time whether the sum of the elements in that subset equals the target number.

Wikipedia Subset sum problem

5.8 Boolean satisfiability problem

Definitions A **boolean formula** consists of boolean (true/false) variables and operators $\wedge$, $\vee$, and $\neg$, with the following (standard) definitions:

x	y	$x \wedge y$	$x \vee y$	$\neg x$
$false$	$false$	$false$	$false$	$true$
$false$	$true$	$false$	$true$	
$true$	$false$	$false$	$true$	$false$
$true$	$true$	$true$	$true$	

A boolean formula is called **satisfiable** when there exists an assignment of boolean values to the boolean variables such that the formula evaluates to $true$.

The **boolean satisfiability problem**, abbreviated as **SAT**, is a decision problem, where an instance consists of a boolean formula and the question is whether this formula is satisfiable.

Properties

- SAT is in class NP. A certificate for a yes-instance is an assignment that assigns a boolean value to each boolean variable in the formula. It can be checked in polynomial time whether the formula evaluates to $true$ under this assignment.

Wikipedia Boolean satisfiability problem

5.9 Polynomial Reduction

Definitions A **polynomial reduction** (§2.5) of problem P to problem Q is an algorithm that transforms (reduces) each instance of P to a (polynomially bounded) number of instances of Q and combines the solutions for the Q-instances into a solution for the P-instance, in polynomial time (ignoring the time to find solutions to the Q-instances). The existence of such a reduction shows that problem P is no harder than problem Q: if you can solve Q, then you also have a solution to P.

Properties

1. The knowledge that a decision problem Q is in class P (§5.5) can be used to prove that another decision problem P is in P as well, by giving a polynomial reduction of problem P to problem Q. By combining the polynomial reduction of P to Q with the polynomial-time algorithm for deciding Q, we obtain a polynomial-time algorithm for deciding P.
2. A common type of polynomial reduction (of problem P to problem Q) is that where a P-instance is reduced to a single Q-instance, whose solution also solves the P-instance. That is, the transformation of the solution of the Q-instance to a solution of the P-instance is the identity transformation (which is polynomial). This is also known as a **Karp reduction**.

Wikipedia Polynomial reduction

5.10 NP-hard, NP-complete

Definitions Informally, decision problem P is **NP-hard** when P is at least as hard as every problem in NP, and more formally, when for each problem Q in class NP there exists a polynomial reduction (§5.9) from Q to P. In other words, a solution to P can be transformed in a solution to Q, with a performance loss that is at most polynomial.

A decision problem is **NP-complete** when it is NP-hard and in class NP.

Properties

1. If there exists a polynomial-time algorithm for any NP-hard problem, then all problems in NP can be solved in polynomial time (cf. Property 1 of polynomial reductions (§5.9)). Consequently, NP would be contained in P, and this would solve the P=NP problem (§5.5) in the affirmative.

2. The knowledge that a decision problem P is NP-hard can be used to prove that another decision problem Q is NP-hard as well, by giving a polynomial reduction of problem P to problem Q.

   Here is why. Assume that P is NP-hard, that is, for each problem R in NP, there exists a polynomial reduction from R to P. Our goal is to prove that Q is NP-hard, that is, that for each problem R in NP, there exists a polynomial reduction from R to Q. To achieve the letter, let R be in NP. Since P is NP-hard, there exists a polynomial reduction from R to P. Furthermore, we assumed that there exists a polynomial reduction from P to Q. Composing these two polynomial reductions yields the desired polynomial reduction from R to Q.

3. TSP, subset sum, and SAT are NP-complete decision problems. SAT serves as the *mother problem* of NP-complete problems: it was the first decision problem proven to be NP-complete, and subsequently other problems were proven NP-complete via polynomial reduction from SAT.

4. There exist NP-hard problems that are *not* in NP. For instance, the halting problem is NP-hard but not in NP. The latter is obvious, since problems in NP are by definition decidable. The former follows from Property 2 by applying it to the subset sum problem, which is NP-complete, hence NP-hard, using a polynomial reduction from the subset sum problem to the halting problem. That polynomial reduction proceeds like this. Consider an instance of the subset sum problem. Construct a Turing machine that iterates over all subsets of the given set and goes into an infinite loop when it encounters a subset that sums to the given target, and if it does not encounter such a subset then it halts. This construction can be done in polynomial time, and indeed shows that the ability to solve the halting problem would give one the ability to solve the subset sum problem. **Extra**

   Note that it does not matter that we cannot actually solve the halting problem. NP-hardness is about the ability to transform problem instances, not about the existence of algorithms that solve the problems involved.

Wikipedia NP-hardness — NP-completeness

5.11 Numerical Limitations

Extra

See [5].

6 Conclusion

In this course, we have explored the fundamental concepts and insights of informatics, summarized below.

Models of computation capture the fundamental nature of computation, where we encountered fundamental boundaries between models in terms of differences in computational power.

- Regular: Finite Automata
- Universal: Turing Machines, Universal Turing Machines, Random Access Machines

Algorithms provide effective and efficient solutions to fundamental computational problems, and how to reason about them.

- Sorting and searching
- Some graph algorithms
- Loop invariants

Information is the fundamental 'stuff' that computations operate on, where we encountered fundamental limits on efficiency, reliability, and security of communication and storage.

- Shannon's Source and Channel Coding Theorems
- Perfect secrecy, symmetric (secret-key) and asymmetric (public-key) cryptosystems, digital signatures

Limits of computation where we encountered fundamental limits to what can be computed, in principle and in practice.

- Decidable versus undecidable problems
- Tractable versus intractable problems
- P versus NP, and NP-completeness

6.1 Suggestions for Further Exploration

- Conway's Game of Life
- Lambda calculus
- DNA computing, molecular computing, quantum computing
- **it from bit**, the universe as a computation
- Algorithmic Information Theory
- Randomness, derandomization
- Approximation algorithms for optimization problems

Acknowledgements

Lecturers: Bas Luttik, Bettina Speckmann, Arthur van Goethem, Tom Verhoeff

Lecture notes: Tom Verhoeff

References

[1] Peter J. Bentley. *Digital Biology*. eBook, 2011. `http://about.peterjbentley.com`, see My Books (accessed 21 Dec 2015).

[2] Thomas H. Cormen. *Algorithms Unlocked*. MIT Press, 2013.

[3] A. K. Dewdney. *The New Turing Omnibus*. Computer Science Press, 1993.

[4] David Harel. *Computers Ltd: What They REALLY Can't Do*. Oxford University Press, 2000, 2003, 2012.

[5] Gyula Horváth and Tom Verhoeff. "Numerical Difficulties in Pre-University Informatics Education and Competitions", *Informatics in Education*, **2**(1):21–38 (2012).

[6] Arefin Huq (Editor). *XRDS: Crossroads, The ACM Magazine for Students — The Legacy of Alan Turing: Pushing the Boundaries of Computation* **18**(3), Spring 2012.

[7] Claude Shannon. "A Mathematical Theory of Communication." *The Bell System Technical Journal*, **27**:379–423, 623–656, July, October, 1948.

[8] Tom Verhoeff. "Informatics Everywhere: Information and Computation in Society, Science, and Technology". *Olympiads in Informatics* **7**:140-152 (2013).

Index