CHAPTER 4

Computers, Numbers and Text

In this chapter we are going to study how numbers are represented in a computer. We already know that at the most basic level, computers just handle sequences of 0s and 1s. We also know that numbers can be represented in different numeral systems, in particular the binary (base-2) numeral system which is perfectly suited for computers. We first consider representation of integers which is quite straightforward, and then representation of fractional numbers which is a bit more challenging.

4.1 Representation of Integers

If computers are to perform calculations with integers, we must obviously have a way to represent the numbers in terms of the computers' electronic circuitry. This presents us with one major challenge and a few minor ones. The big challenge is that integer numbers can become arbitrarily large in magnitude. This means that there is no limit to the number of digits that may be needed to write down integer numbers. On the other hand, the resources in terms of storage capacity and available computing time is always finite, so there will always be an upper limit on the magnitude of numbers that can be handled by a given computer. There are two standard ways to handle this problem.

The most common solution is to restrict the number of digits. If for simplicity we assume that we can work in the decimal numeral system, we could restrict the number of digits to 6. This means that the biggest number we can handle would be 999999. The advantage of this limitation is that we could put a lot of effort into making the computer's operation on 6 digit decimal numbers

Traditional		SI prefixes			
Symbol	Value	Symbol	Value	Alternative	Value
kB (kilobyte)	2^{10}	KB	10^3	kibibyte	2^{10}
MB (megabyte)	2^{20}	MB	10^6	mibibyte	2^{20}
GB (gigabyte)	2^{30}	GB	10^9	gibibyte	2^{30}
TB (terabyte)	2^{40}	TB	10^{12}	tibibyte	2^{40}
PB (petabyte)	2^{50}	PB	10^{15}	pibibyte	2^{50}
EB (exabyte)	2^{60}	EB	10^{18}	exbibyte	2^{60}
ZB (zettabyte)	2^{70}	ZB	10^{21}	zebibyte	2^{70}
YB (yottabyte)	2^{80}	YB	10^{24}	yobibyte	2^{80}

Table 4.1. The Si-prefixes for large collections of bits and bytes.

extremely efficient. On the other hand the computer could not do much other than report an error message and give up if the result should become larger than 999999.

The other solution would be to not impose a specific limit on the size of the numbers, but rather attempt to handle as large numbers as possible. For any given computer there is bound to be an upper limit, and if this is exceeded the only response would be an error message. We will discuss both of these approaches to the challenge of big numbers below.

4.1.1 Bits, bytes and numbers

At the most basic level, the circuitry in a computer (usually referred to as the *hardware*) can really only differentiate between two different states, namely '0' and '1' (or 'false' and 'true'). This means that numbers must be represented in terms of 0 and 1, in other words in the binary numeral system. From what we learnt in the previous chapter, this is not a difficult task, but for reasons of efficiency the electronics have been designed to handle groups of binary digits. The smallest such group consists of 8 binary digits (bits) and is called a byte. Larger groups of bits are usually groups of bytes. For manipulation of numbers, groups of 4 and 8 bytes are usually used, and computers have special computational units to handle groups of bits of these sizes.

Fact 4.1. *A binary digit is called a bit and a group of 8 bits is called a byte. Numbers are usually represented in terms of 4 bytes (32 bits) or 8 bytes (64 bits).*

The standard SI prefixes are used when large amounts of bits and bytes are referred to, see Table 4.1. Note that traditionally the factor between each prefix has been $1024 = 2^{10}$ in the computer world, but use of the SI-units is now encouraged. However, memory size is always reported using the traditional binary units and most operating systems also use these units to report hard disk sizes and file sizes. So a file containing 3 913 880 bytes will typically be reported as being 3.7 MB.

To illustrate the size of the numbers in Table 4.1 it is believed that the world's total storage in 2006 was 160 exabytes, and the projection is that this will grow to nearly one zettabyte by 2010.

4.1.2 Fixed size integers

Since the hardware can handle groups of 4 or 8 bytes efficiently, the representation of integers are usually adapted to this format. If we use 4 bytes we have 32 binary digits at our disposal, but how should we use these bits? We would certainly like to be able to handle both negative and positive numbers, so we use one bit to signify whether the number is positive or negative. We then have 31 bits left to represent the binary digits of the integer. This means that the largest 32-bit integer that can be handled is the number where all 31 digits are 1, i.e.,

$$1 \cdot 2^{30} + 1 \cdots 2^{29} + \cdots + 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = 2^{31} - 1.$$

Based on this it may come as a little surprise that the most negative number that can be represented is -2^{31} and not $-2^{31} + 1$. The reason is that with 32 bits at our disposal we can represent a total of 2^{32} numbers. Since we need 2^{31} bit combinations for the positive numbers and 0, we have $2^{32} - 2^{31} = 2^{31}$ combinations of digits left for the negative numbers. Similar limits can be derived for 64-bit integers.

Fact 4.2. *The smallest and largest numbers that can be represented by 32-bit integers are*

$$I_{\min 32} = -2^{31} = -2147483648, \quad I_{\max 32} = 2^{31} - 1 = 2147483647.$$

With 64-bit integers the corresponding numbers are

$$I_{\min 64} = -2^{63} = -9223372036854775808,$$
$$I_{\max 64} = 2^{63} - 1 = 9223372036854775807.$$

What we have discussed so far is the typical hardware support for integer numbers. When we program a computer we have to use a suitable programming language, and different languages may provide different interfaces to the hardware. There are a myriad of computer languages, and especially handling of integers may differ quite a bit. We will briefly review integer handling in two languages, Java and Python, as representatives of two different approaches.

4.1.3 Integers in Java

Java is a typed language which means that the type of all variables has to be stated explicitly. If we wish to store 32-bit integers in the variable `n`, we use the declaration `int n` and we say that `n` is an `int` variable. If we wish to use `n` as a 64-bit variable, we use the declaration `long n` and say that `n` is a `long` variable. Integers appearing to the right of an assignment are considered to be of type `int`, but you may specify that an integer is to be interpreted as a `long` integer by appending an `L`. In other words, an expression like `2+3` will be computed as an `int` whereas the expression `2L+3L` will be computed as a `long`, using 64 bits.

Since Java has integer types of fixed size, something magic must happen when the result of an integer computation becomes too large for the type. Suppose for example that we run the code segment

```
int a;
a = 2147483647;
a = a + 1;
```

The staring value for `a` is the largest possible 32-bit integer, and when we add `1` we obtain a number that is too big for an `int`. This is referred to by saying that an *overflow* occurs. So what happens when an integer overflows in Java? The statements above will lead to `a` receiving the value `-2147483648`, and Java gives no warning about this strange behaviour!. If you look carefully the result is -2^{31}, i.e., the smallest possible `int`. Basically Java (and similar languages) consider the 32-bit integers to lie in a ring where the integer succeeding $2^{31}-1$ is -2^{31} (overflow in `long` integers are handled similarly). Sometimes this may be what you want, but most of the time this kind of behaviour is probably going to give you a headache unless you remember this paragraph!

Note that Java also has 8 bit integers (`byte`) and 16 bit integers (`short`). These behave completely analogously to `int` and `long` variables.

It is possible to work with integers that require more than 64 bits in Java, but then you have to resort to an auxiliary class called `BigInteger`. In this class integers are only limited by the total resources available on your computer, but the cost of resorting to `BigInteger` is a big penalty in terms of computing time.

4.1.4 Integers in Python

Python handles integers quite differently from Java. First of all you do not need to declare the type of variables in Python. So if you write something like `a=2+3` then Python will look at the right-hand side of the assignment, conclude that this is an integer expression and store the result in an integer variable. An integer variable in Python is called an `int` and on most computers this will be a 32-bit integer. The good news is that Python handles overflow much more gracefully than Java. If Python encounters an integer expression that is greater than $2^{31}-1$ it will be converted to what is called a `long` integer variable in Python. Such variables are only bounded by the available resources on the computer, just like `BigInteger` in Java. You can force an integer expression that fits into an `int` to be treated as a `long` integer by using the function `long`. For example, the expression `long(13)` will give the result `13L`, i.e., the number 13 represented as a `long` integer. Similarly, the expression `int(13L)` will convert back to an `int`.

This means that overflow is very seldom a problem in Python, as virtually all computers today should have sufficient resources to avoid overflow in ordinary computations. But it may of course happen that you make a mistake that result in a computation producing very large integers. You will notice this in that your program takes a very long time and may seem to be stuck. This is because your computation is consuming all resources in the computer so that everything else comes to a standstill. You could wait until you get an error message, but this make take a long time so it is usually better to just abort the computation.

Since `long` integers in Python can become very large, it may be tempting to use them all the time and ignore the `int` integers. The problem with this is that the `long` integers are implemented in extra program code (usually referred to as *software*), just like the `BigInteger` type in Java, and this is comparatively slow. In contrast, operations with `int` integers are explicitly supported by the hardware and is very fast.

4.1.5 Division by zero

Other than overflow, the only potential problem with integer computation is division by zero. This is mathematically illegal and results in an error message and the computations being halted (or an exception is raised) in all programming languages.

4.2 Computers and real numbers

Computations with integers is not sufficient for many parts of mathematics; we must also be able to compute with real numbers. And just like for integers, we want fast computations so we can solve large and challenging problems. This

inevitably means that there will be limitations on the class of real numbers that can be handled efficiently by computers.

To illustrate the challenge, consider the two real numbers

$$\pi = 3.14159265358979323846264338327950288419 7\ldots,$$
$$10^6\pi = 3.14159265358979323846264338327950288419 7\ldots \times 10^6.$$

Both of these numbers are irrational and require infinitely many digits in any numeral system with an integer base. With a fixed number of digits at our disposal we can only store the most significant (the left-most) digits, which means that we have to ignore infinitely many digits. But this is not enough to distinguish between the two numbers π and $10^6\pi$, we also have to store information about the size of the numbers.

The fact that many real numbers have infinitely many digits and we can only store a finite number of these means that there is bound to be an error when real numbers are represented on a computer. This is in marked contrast to integer numbers where there is no error, just a limit on the size of numbers. The errors are usually referred to as *rounding errors* or *round-off errors*. These errors are also present on calculators and a simple situation where round-off error can be observed is by computing $\sqrt{2}$, squaring the result and subtracting 2. On one calculator the result is approximately 4.4×10^{-16}, a clear manifestation of round-off error.

Usually the round-off error is small and remains small throughout a computation. In some cases however, the error grows throughout a computation and may become significant. In fact, there are situations where the round-off error in a result is so large that all the displayed digits are wrong! Computations which lead to large round-off errors are said to be *badly conditioned* while computations with small errors are said to be *well conditioned*.

Since some computations may lead to large errors it is clearly important to know in advance if a computation may be problematic. Suppose for example you are working on the development of a new aircraft and you are responsible for simulations of the forces acting on the wings during flight. Before the first flight of the aircraft you had better be certain that the round-off errors (and other errors) are under control. Such error analysis is part of the field called *Numerical Analysis*.

4.2.1 Representation of real numbers

To understand round-off errors and other characteristics of how computers handle real numbers, we must understand how real numbers are represented. We

are going to do this by first pretending that computers work in the decimal numeral system. Afterwards we will translate our observations to the binary representation that is used in practice.

Any real number can be expressed in the decimal system, but infinitely many digits may be needed. To represent such numbers with finite resources we must limit the number of digits. Suppose for example that we use four decimal digits to represent real numbers. Then the best representations of the numbers π, $1/700$ and $100003/17$ would be

$$\begin{aligned} \pi &\approx 3.142, \\ \frac{1}{700} &\approx 0.001429, \\ \frac{100003}{17} &\approx 5883. \end{aligned}$$

If we consider the number $100000000/23 \approx 4347826$ we see that it is not representable with just four digits. However, if we write the number as 0.4348×10^7 we can represent the number if we also store the exponent 7. This is the background for the following simple observation.

Observation 4.1 (Normal form of real number)**.** *Let a be a real number different from zero. Then a can be written uniquely as*

$$a = b \times 10^n \tag{4.1}$$

where b is bounded by

$$\frac{1}{10} \leq |b| < 1 \tag{4.2}$$

and n is an integer. This is called the normal form of a, and the number b is called the significand while n is called the exponent of a. The normal form of 0 is $0 = 0 \times 10^0$.

Note that the digits of a and b are the same; to arrive at the normal form in (4.1) we simply multiply a by the power of 10 that brings b into the range given by (4.2).

The normal form of π, $1/7$, $100003/17$ and $10000000/23$ are

$$\begin{aligned} \pi &\approx 0.3142 \times 10^{1}, \\ \frac{1}{7} &\approx 0.1429 \times 10^{-2}, \\ \frac{100003}{17} &\approx 0.5883 \times 10^{4}, \\ \frac{10000000}{23} &\approx 0.4348 \times 10^{7}. \end{aligned}$$

From this we see that if we reserve four digits for the significand and one digit for the exponent, plus a sign for both, then we have a format that can accommodate all these numbers. If we keep the significand fixed and vary the exponent, the decimal point moves among the digits. For this reason this kind of format is called *floating point*, and numbers represented in this way are called *floating point numbers*.

It is always useful to be aware of the smallest and largest numbers that can be represented in a format. With four digits for the significand and one digit for the exponent plus signs, these numbers are

$$\begin{aligned} -0.9999 \times 10^{9}, \quad & -0.1000 \times 10^{-9}, \\ 0.1000 \times 10^{-9}, \quad & 0.9999 \times 10^{9}. \end{aligned}$$

In practice, a computer uses a binary representation. Before we consider details of how many bits to use etc., we must define a normal form for binary numbers. This is a straightforward generalisation from the decimal case.

Observation 4.2 (Binary normal form of real number)**.** *Let a be a real number different from zero. Then a can be written uniquely as*

$$a = b \times 2^{n}$$

where b is bounded by

$$\frac{1}{2} \leq |b| < 1$$

and n is an integer. This is called the binary normal form of a, and the number b is called the significand while n is called the exponent of a. The normal form of 0 is $0 = 0 \times 2^{0}$.

This is completely analogous to the decimal version in Observation 4.1 in that all occurrences of 10 have been replaced by 2. Most of today's computers

use 32 or 64 bits to represent real numbers. The 32-bit format is useful for applications that do not demand very much accuracy, but 64 bits has become a standard for most scientific applications. Occasionally higher accuracy is required in which case there are some formats with more bits or even a format with no limitation other than the resources available in the computer.

To describe a floating point format, it is not sufficient to state how many bits are used in total, we also have to know how many bits are used for the significand and how many for the exponent. There are several possible ways to do this, but there is an international standard for floating point computations that is used by most computer manufacturers. This standard is referred to as the IEEE[1] 754 standard, and the main details of the 32-bit version is given below.

Fact 4.3 (IEEE 32-bit floating point format)**.** *With 32-bit floating point numbers 23 bits are allocated for the significand and 9 bits for the exponent, both including signs. This means that numbers have about 6–9 significant decimal digits. The smallest and largest negative numbers in this format are*

$$F^-_{\min 32} \approx -3.4 \times 10^{38}, \quad F^-_{\max 32} \approx -1.4 \times 10^{-45}.$$

The smallest and largest positive numbers are

$$F^+_{\min 32} \approx 1.4 \times 10^{-45}, \quad F^+_{\max 32} \approx 3.4 \times 10^{38}.$$

This is just a summary of the most important characteristics of the 32-bit IEEE-standard; there are a number of details that we do not want to delve into here. However, it is worth pointing out that when any nonzero number a is expressed in binary normal form, the first bit of the significand will always be 1 (remember that we simply shift the binary point until the first bit is 1). Since this bit is always 1, it does not need to be stored. This means that in reality we have 24 bits (including sign) available for the significand. The only exception to this rule is when the exponent has its smallest possible value. Then the first bit is assumed to be 0 (these correspond to so-called denormalised numbers) and this makes it possible to represent slightly smaller numbers than would otherwise be possible. In fact the smallest positive 32-bit number with 1 as first bit is approximately 1.2×10^{-38}.

Not all bit combinations in the IEEE standard are used for ordinary numbers. Three of the extra 'numbers' are `-Infinity`, `Infinity` and `NaN`. The infinities

[1] IEEE is an abbreviation for Institute of Electrical and Electronics Engineers which is a professional technological association.

typically occur during overflow. For example, if you use 32-bit floating point and perform the multiplication $10^{30} * 10^{30}$, the result will be `Infinity`. The negative infinity behaves in a similar way. The `NaN` is short for 'Not a Number' and is the result if you try to perform an illegal operation. A typical example is if you try to compute $\sqrt{-1}$ without using complex numbers, this will give `NaN` as the result. And once you have obtained a `NaN` result it will pollute anything that it touches; `NaN` combined with anything else will result in `NaN`.

With 64-bit numbers we have 32 extra bits at our disposal and the question is how these should be used. The creators of the IEEE standard believed improved accuracy to be more important than support for very large or very small numbers. They therefore increased the number of bits in the significand by 30 and the number of bits in the exponent by 2.

Fact 4.4 (IEEE 64-bit floating point format)**.** *With 64-bit floating point numbers 53 bits are allocated for the significand and 11 bits for the exponent, both including signs. This means that numbers have about 15–17 significant decimal digits. The smallest and largest negative number in this format are*

$$F^{-}_{\min 64} \approx -1.8 \times 10^{308}, \quad F^{-}_{\max 64} \approx -5 \times 10^{-324}.$$

The smallest and largest positive numbers are

$$F^{+}_{\min 64} \approx 5 \times 10^{-324}, \quad F^{+}_{\max 64} \approx 1.8 \times 10^{308}.$$

Other than the extra bits available, the 64-bit format behaves just like its 32-bit little brother, with the leading 1 not being stored, the use of denormalised numbers, `-Infinity`, `Infinity` and `NaN`.

4.2.2 Floating point numbers in Java

Java has two floating point types, `float` and `double`, which are direct implementations of the 32-bit and 64-bit IEEE formats described above. In Java the result of `1.0/0.0` will be `Infinity` without a warning.

4.2.3 Floating point numbers in Python

In Python floating point numbers come into action as soon as you enter a number with a decimal point. Such numbers are represented in the 64-bit format described above and most of the time the computations adhere to the IEEE standard. However, there are some exceptions. For example, the division `1.0/0.0` will give an error message and the symbol for 'Infinity' is `Inf`.

In standard Python, there is no support for 32-bit floating point numbers. However, you gain access to this if you import the `NumPy` library.

4.3 Representation of letters and other characters

At the lowest level, computers can just handle 0s and 1s, and since any number can be expressed uniquely in the binary number system it can also be represented in a computer (except for the fact that we may have to limit both the size of the numbers and their number of digits). We all know that computers can also handle text and in this section we are going to see the basic principles of how this is done.

A text is just a sequence of individual characters like 'a', 'B', '3', '.', '?', i.e., upper- and lowercase letters, the digits 0–9 and various other symbols used for punctuation and other purposes. So the basic challenge in handling text is how to represent the individual characters. With numbers at our disposal, this is a simple challenge to overcome. Internally in the computer a character is just represented by a number and the correspondence between numbers and characters is stored in a table. The letter 'a' for example, usually has code 97. So when the computer is told to print the character with code 97, it will call a program that draws an 'a'[2]. Similarly, when the user presses the 'a' on the keyboard, it is immediately converted to code 97.

Fact 4.5 (Representation of characters)**.** *In computers, characters are represented in terms of integer codes and a table that maps the integer codes to the different characters. During input each character is converted to the corresponding integer code, and during output the code is used to determine which character to draw.*

Although the two concepts are slightly different, we will use the terms 'character sets' and 'character mappings' as synonyms.

From Fact 4.5 we see that the character mapping is crucial in how text is handled. Initially, the mappings were simple and computers could only handle the most common characters used in English. Today there are extensive mappings available that make the characters of most of the world's languages, including the ancient ones, accessible. Below we will briefly describe some of the most common character sets.

4.3.1 The ASCII Table

In the infancy of the digital computer there was no universal standard for mapping characters to numbers. This made it difficult to transfer information from one computer to another, and the need for a standard soon became apparent. The first version of ASCII (American Standard Code for Information Interchange) was published in 1963 and it was last updated in 1986. ASCII defines codes for 128 characters that are commonly used in English plus some more technical characters. The fact that there are $128 = 2^7$ characters in the ASCII table means that 7 bits are needed to represent the codes. Today's computers usually handle one byte (eight bits) at a time so the ASCII character set is now normally just part of a larger character set, see below.

Table 4.2 (towards the end of this chapter) shows the ASCII characters with codes 32–127. We notice the upper case letters with codes 65–90, the lower case letters with codes 97–122 and the digits 0–9 with codes 48–57. Otherwise there are a number of punctuation characters and brackets as well as various other characters that are used more or less often. Observe that there are no characters from the many national alphabets that are used around the world. ASCII was developed in the US and was primarily intended to be used for giving a textual representation of computer programs which mainly use vocabulary from English. Since then computers have become universal tools that process all kinds of information, including text in many different languages. As a result new character sets have been developed, but almost all of them contain ASCII as a subset.

Character codes are used for arranging words in alphabetical order. To compare the two words 'high' and 'all' we just check the character codes. We see that 'h' has code 104 while 'a' has code 97. So by ordering the letters according to their character codes we obtain the normal alphabetical order. Note that the codes of upper case letters are smaller than the codes of lower case letters. This means that capitalised words and words in upper case precede words in lower case in the standard ordering.

Table 4.3 shows the first 32 ASCII characters. These are quite different from most of the others (with the exception of characters 32 and 127) and are called *control characters*. They are not intended to be printed in ink on paper, but rather indicate some kind of operation to be performed by the printing equipment or a signal to be communicated to a sender or receiver of the text. Some of the characters are hardly used any more, but others have retained their significance. Character 4 (^D) has the description 'End of Transmission' and is often used to signify the end of a file, at least under Unix-like operating systems. Because of this, many programs that operate on files, like for example text-editors,

[2] The shape of the different characters are usually defined as mathematical curves.

will quit if you type ^D (hold down the control-key while you press 'd'). Various combinations of characters 10, 12 and 13 are used in different operating systems for indicating a new line within a file. The meaning of character 13 ('Carriage Return') was originally to move back to the beginning of the current line and character 10 ('Line Feed') meant forward one line.

4.3.2 ISO latin character sets

As text processing by computer became important in the 1980s, extensions of the ASCII character set that included various national characters used in European languages were needed. The International Standards Organisation (ISO) developed a number of such character sets, like ISO Latin 1 ('Western'), ISO Latin 2 ('Central European') and ISO Latin 5 ('Turkish'), and so did several computer manufacturers. Virtually all of these character sets retained ASCII in the first 128 positions, but increased the code from seven to eight bits to accomodate another 128 characters. This meant that different parts of the Western world had local character sets which could encode their national characters, but if a file was interpreted with the wrong character set, some of the characters beyond position 127 would come out wrong.

Table 4.4 shows characters 192–255 in the ISO Latin 1 character set. These include most latin letters with diacritics used in the Western European languages. Positions 128–191 in the character set are occupied by some control characters similar to those at the beginning of the ASCII table but also a number of other useful characters.

4.3.3 Unicode

By the early 1990s there was a critical need for character sets that could handle multilingual characters, like those from English and Chinese, in the same document. A number of computer companies therefore set up an organisation called Unicode. Unicode has since then organised the characters of most of the world's languages in a large table called the Unicode table, and new characters are still being added. There are a total of about 100 000 characters in the table which means that at least three bytes are needed for their representation. The codes range from 0 to 1114111 (hexadecimal $10ffff_{16}$) which means that only about 10 % of the table is filled. The characters are grouped together according to language family or application area, and the empty spaces make it easy to add new characters that may come into use. The first 256 characters of Unicode is identical to the ISO Latin 1 character set, and in particular the first 128 characters correspond to the ASCII table. You can find all the Unicode characters at `http://www.unicode.org/charts/`.

One could use the same strategy with Unicode as with ASCII and ISO Latin 1 and represent the characters via their integer codes (usually referred to as *code points*) in the Unicode table. This would mean that each character would require three bytes of storage. The main disadvantage of this is that a program for reading Unicode text would give completely wrong results if by mistake it was used for reading 'old fashioned' eight bit text, even if it just contained ASCII characters. Unicode has therefore developed variable length encoding schemes for encoding the characters.

4.3.4 UTF-8

A popular encoding of Unicode is UTF-8[3]. UTF-8 has the advantage that ASCII characters are encoded in one byte so there is complete backwards compatibility with ASCII. All other characters require from two to four bytes.

To see how the code points are actually encoded in UTF-8, recall that the ASCII characters have code points in the range 0–127 (decimal) which is 00_{16}–$7f_{16}$ in hexadecimal or 00000000_2–01111111_2 in binary. These characters are just encoded in one byte in the obvious way and are characterised by the fact that the most significant (the left-most) bit is 0. All other characters require more than one byte, but the encoding is done in such a way that none of these bytes start with 0. This is done by adding some set fixed bit combinations at the beginning of each byte. Such codes are called prefix codes. The details are given in a fact box.

Fact 4.6 (UTF-8 encoding of Unicode)**.** *A Unicode character with code point c is encoded in UTF-8 according to the following four rules:*

1. *If $c = (d_6d_5d_4d_3d_2d_1d_0)_2$ is in the decimal range 0–127 (hexadecimal 00_{16}–$7f_{16}$), it is encoded in one byte as*

$$0d_6d_5d_4d_3d_2d_1d_0.$$

2. *If $c = (d_{10}d_9d_8d_7d_6d_5d_4d_3d_2d_1d_0)_2$ is in the decimal range 128–2047 (hexadecimal 80_{16}–$7ff_{16}$) it is encoded as the two byte binary number*

$$110d_{10}d_9d_8d_7d_6 \quad 10d_5d_4d_3d_2d_1d_0.$$

3. *If $c = (d_{15}d_{14}d_{13}d_{12}d_{11}d_{10}d_9d_8d_7d_6d_5d_4d_3d_2d_1d_0)_2$ is in the decimal range 2048–65535 (hexadecimal 800_{16}–$ffff_{16}$) it is encoded as the three byte binary number*

$$1110d_{15}d_{14}d_{13}d_{12} \quad 10d_{11}d_{10}d_9d_8d_7d_6 \quad 10d_5d_4d_3d_2d_1d_0.$$

[3] UTF is an abbreviation of Unicode Transformation Format.

4. *If* $c = (d_{20}d_{19}d_{18}d_{17}d_{16}d_{15}d_{14}d_{13}d_{12}d_{11}d_{10}d_9d_8d_7d_6d_5d_4d_3d_2d_1d_0)_2$ *is in the decimal range 65536–1114111 (hexadecimal* 10000_{16}*–*$10ffff_{16}$*) it is encoded as the four byte binary number*

$$11110d_{20}d_{19}d_{18} \quad 10d_{17}d_{16}d_{15}d_{14}d_{13}d_{12} \quad 10d_{11}d_{10}d_9d_8d_7d_6 \quad 10d_5d_4d_3d_2d_1d_0.$$

This may seem complicated at first sight, but is in fact quite simple and elegant. Note any given byte in a UTF-8 encoded text must start with the binary digits 0, 10, 110, 1110 or 11110. If the first bit in a byte is 0, the remaining bits represent a seven bit ASCII character. If the first two bits are 10, the byte is the second, third or fourth byte of a multi-byte code point, and we can find the first byte by going back in the byte stream until we find a byte that does not start with 10. If the byte starts with 110 we know that it is the first byte of a two-byte code point; if it starts with 1110 it is the first byte of a three-byte code point; and if it starts with 11110 it is the first of a four-byte code point.

Observation 4.3. *It is always possible to tell if a given byte within a text encoded in UTF-8 is the first, second, third or fourth byte in the encoding of a code point.*

The UTF-8 encoding is particularly popular in the Western world since all the common characters of English can be represented by one byte, and almost all the national European characters can be represented with two bytes.

4.3.5 UTF-16

Another common encoding is UTF-16. In this encoding most Unicode characters with two-byte code points are encoded directly by their code points. Since the characters of major Asian languages like Chinese, Japanese and Korean are encoded in this part of Unicode, UTF-16 is popular in this part of the world. UTF-16 is also the native format for representation of text in the recent versions of Microsoft Windows and Apple's Mac OS X as well as in programming environments like Java, .Net and Qt.

UTF-16 uses a variable width encoding scheme similar to UTF-8, but the basic unit is two bytes rather than one. This means that all code points are encoded in two or four bytes. In order to recognize whether two consecutive bytes in an UTF-16 encoded text correspond to a two-byte code point or a four-byte code

point, the initial bit patterns of each pair of a four byte code has to be illegal in a two-byte code. This is possible since there are big gaps in the Unicode table. In fact certain Unicode code points are reserved for the specific purpose of signifying the start of pairs of four-byte codes (so-called surrogate pairs).

Fact 4.7 (UTF-16 encoding of Unicode)**.** *A Unicode character with code point c is encoded in UTF-16 according to two rules:*

1. *If $c = (d_{15}d_{14}d_{13}d_{12}d_{11}d_{10}d_9d_8d_7d_6d_5d_4d_3d_2d_1d_0)_2$ is a code point in the range 0–65535 (hexadecimal 0000_{16}–$ffff_{16}$) it is encoded as the two bytes*

$$d_{15}d_{14}d_{13}d_{12}d_{11}d_{10}d_9d_8 \quad d_7d_6d_5d_4d_3d_2d_1d_0.$$

2. *If $c = (d_{20}d_{19}d_{18}d_{17}d_{16}d_{15}d_{14}d_{13}d_{12}d_{11}d_{10}d_9d_8d_7d_6d_5d_4d_3d_2d_1d_0)_2$ is a code point in the decimal range 65536–1114111 (hexadecimal 10000_{16}–$10ffff_{16}$), compute the number $c' = c - 65536$ (subtract 10000_{16}). This number can be represented by 20 bits,*

$$c' = (d'_{19}d'_{18}d'_{17}d'_{16}d'_{15}d'_{14}d'_{13}d'_{12}d'_{11}d'_{10}d'_9d'_8d'_7d'_6d'_5d'_4d'_3d'_2d'_1d'_0)_2.$$

The encoding of c is then given by the four bytes

$$110110d'_{19}d'_{18} \quad d'_{17}d'_{16}d'_{15}d'_{14}d'_{13}d'_{12}d'_{11}d'_{10}$$
$$110111d'_9d'_8 \quad d'_7d'_6d'_5d'_4d'_3d'_2d'_1d'_0.$$

Superficially it may seem like UTF-16 does not have the prefix property, i.e., it may seem that a pair of bytes produced by rule 2 may occur as a pair generated by rule 1 and vice versa. However, the existence of gaps in the Unicode table means that this problem does not occur.

Observation 4.4. *None of the pairs of bytes produced by rule 2 in Fact 4.7 will ever match a pair of bytes produced by the first rule as there are no two-byte code points that start with the bit sequences 110110 or 110111. It is therefore always possible to determine whether a given pair of consecutive bytes in an UTF-16 encoded text corresponds directly to a code point (rule 1), or is the first or second pair of a four byte encoding.*

The UTF-16 encoding has the advantage that all two-byte code points are encoded directly by their code points. Since the characters that require more

than two-byte code points are very rare, this means that virtually all characters are encoded directly in two bytes.

UTF-16 has one technical complication. Different computer architectures code pairs of bytes in different ways: Some will insist on sending the eight most significant bits first, some will send the eight least significant bits first; this is usually referred to as *little endian* and *big endian*. To account for this there are in fact three different UTF-16 encoding schemes, UTF-16, UTF-16BE and UTF-16LE. UTF-16BE uses strict big endian encoding while UTF-16LE uses strict little endian encoding. UTF-16 does not use a specific endian convention. Instead any file encoded with UTF-16 should indicate the endian by having as its first two bytes what is called a *Byte Order Mark* (BOM). This should be the hexadecimal sequence feff_{16} for big-endian and fffe_{16} for little-endian. This character, which has code point feff, is chosen because it should never legitimately appear at the beginning of a text.

4.3.6 UTF-32

UTF-32 encode Unicode characters by encoding the code point directly in four bytes or 32 bits. In other words it is a fixed length encoding. The disadvantage is that this encoding is rather extravagant since many frequently occurring characters in Western languages can be encoded with only one byte, and almost all characters can be encoded with two bytes. For this reason UTF-32 is little used in practice.

4.3.7 Text in Java

Characters in Java are represented with the `char` data type. The representation is based on the UTF-16 encoding of Unicode so all the Unicode characters are available. The fact that some characters require three bytes to represent their code points is a complication, but this is handled nicely in the libraries for processing text.

4.3.8 Text in Python

Python also has support for Unicode. You can use Unicode text in your source file by including a line which indicates the specific encoding, for example as in

```
# coding=utf-8/
```

You can then use Unicode in your string constants which in this case will be encoded in UTF-8. All the standard string functions also work for Unicode strings, but note that the default encoding is ASCII.

4.4 Representation of general information

So far we have seen how numbers and characters can be represented in terms of bits and bytes. This is the basis for representing all kinds of different information. Let us start with general text files.

4.4.1 Text

A text is simply a sequence of characters. We know that a character is represented by an integer code so a text file is a sequence of integer codes. If we use the ISO Latin 1 encoding, a file with the text

Knut
Mørken

is represented by the hexadecimal codes (recall that each code is a byte)

4b 6e 75 74 0a 4d f8 72 6b 65 6e

The first four bytes you will find in Table 4.2 as the codes for 'K', 'n', 'u' and 't' (remember that the codes of latin characters in ISO Latin 1 are the same as in ASCII). The fifth character has decimal code 10 which you find in Table 4.3. This is the Line feed character which causes a new line on my computer. The remaining codes can all be found in Table 4.2 except for the seventh which has decimal code 248. This is located in the upper 128 ISO Latin 1 characters and corresponds to the Norwegian letter 'ø' as can be seen in Table 4.4.

If instead the text is represented in UTF-8, we obtain the bytes

4b 6e 75 74 0a 4d c3 b8 72 6b 65 6e

We see that these are the same as for ISO Latin 1 except that 'f8' has become two bytes 'c3 b8' which is the two-byte code for 'ø' in UTF-8.

In UTF-16 the text is represented by the codes

ff fe 4b 00 6e 00 75 00 74 00 0a 00 4d 00 f8 00 72 00 6b 00 65 00 6e 00

All the characters can be represented by two bytes and the leading byte is '00' since we only have ISO Latin 1 characters. It may seem a bit strange that the zero byte comes after the nonzero byte, but this is because the computer uses little endian representation. A program reading this file would detect this from the first two bytes which is the byte-order mark referred to above.

4.4.2 Numbers

A number can be stored in a file by finding its binary representation and storing the bits in the appropriate number of bytes. The number $13 = 1101_2$ for example could be stored as a 32 bit integer by storing the the bytes 00 00 00 0d (in hexadecimal).[4] But here there is a possibly confusing point: Why can we not just store the number as a text? This is certainly possible and if we use UTF-8 we can store 13 as the two bytes 31 33 (in hexadecimal). This even takes up less space than the four bytes required by the true integer format. For bigger numbers however the situation is the opposite: Even the largest 32-bit integer can be represented by four bytes if we use integer format, but since it is a ten-digit number we would need ten bytes to store it as a text.

In general it is advisable to store numbers in the appropriate number format (integer or floating point) when we have a large collection of them. This will usually require less space and we will not need to first read a text and then extract the numbers from the text. The advantage of storing numbers as text is that the file can then be viewed in a normal text editor which for example may be useful for debugging.

4.4.3 General information

Bigger computer programs process information that consists of a mixture of numbers and text. Consider for example digital music. For a given song we may like to store its name, artist, lyrics and all the sound data. The first three items of information is conveniently stored as text. As we shall see later, the sound data is just a very long list of numbers. If the music is in CD-format, the numbers are 16-bit integers, i.e., integers in the interval$[-2^{15}, 2^{15}-1]$ or $[-32768, 32767]$, and there are 5.3 million numbers for each minute of music. These numbers can be saved in text format which would require five bytes for most numbers. We can reduce the storage requirement considerably by saving them in standard binary integer format. This format only requires two bytes for each number so the size of the file would be reduced by a factor of almost 2.5.

4.4.4 Computer programs

Since computers only can interpret sequences of 0s and 1s, computer programs must also be represented in this form at the lowest level. All computers come with an *assembly or machine language* which is the level just above the 0s and 1s. Programs written in higher level languages must be translated (compiled) into assembly language before they can be executed. To do regular programming in assembly language is rather tedious and prone to error as many details

[4] For technical reasons integers are in fact usually stored in so-called two's complement.

that happen behind the scenes in high level languages must be programmed in detail. Each command in the assembly language is represented by an appropriate number of bytes, usually four or eight and therefore corresponds to a specific sequence of 0s and 1s.

4.5 A fundamental principle of computing

In this chapter we have seen that by combining bits into bytes, both numbers, text and more general information can be represented, manipulated and stored in a computer. It is important to remember though, that however complex the information, if it is to be processed by computer it must be encoded into a sequence of 0s and 1s. When we want the computer to do anything with the information it must interpret and assemble the bits in the correct way before it can perform the desired operation. Suppose for example that as part of a programming project you need to temporary store some information in a file, for example a sound file in the simple format outlined in Subsection 4.4.3. When you read the information back from the file it is your responsibility to interpret the information in the correct way. In the sound example this means that you must be able to extract the name of the song, the artist, the lyrics and the sound data from the file. One way to do this is to use a special character, that is not otherwise in use, to indicate the end of one field and the beginning of the next. In the first three fields we can allow text of any length while in the last field only 16 bit integers are allowed. This is a simple example of a *file format*, i.e., a precise description of how information is stored. If your program is going to read information from a file, you need to know the file format to read the information correctly.

In many situations well established conventions will tell you the file format. One type of convention is that filenames often end with a dot and three or more characters that identify the file format. Some examples are `.doc` (Microsoft Word), `.html` (web-documents), `.mp3`(mp3-music files), `.jpg` (photos in jpeg-format). If you are going to write a program that will process one of these file formats you have at least two options: You can find a description of the format and write your own functions that read the information from the file, or you can find a software library written by somebody else that has functions for reading the appropriate file format and converting the information to text and numbers that is returned to your program.

Program code is a different type of file format. A programming language has a precise syntax, and specialised programs called *compilers* or *interpreters* translate programs written according to this syntax into low level commands that the computer can execute.

This discussion of file formats illustrates a fundamental principle in computing: A computer must always be told exactly what to do, or equivalently, must know how to interpret the 0s and 1s it encounters.

Fact 4.8 (A principle of computing)**.** *For a computer to function properly it must always be known how it should interpret the 0s and 1s it encounters.*

This principle is absolute, but there are of course many ways to instruct a computer how information should be interpreted. A lot of the interpretation is programmed into the computer via the operating system, programs that are installed on the computer contain code for encoding and decoding information specific to each program, sometimes the user has to tell a given program how to interpret information (for example tell a program the format of a file), sometimes a program can determine the format by looking for special bit sequences (like the endian convention used in a UTF-16 encoded file). And if you write programs yourself you must of course make sure that your program can process the information from a user in an adequate way.

Exercises

4.1 Determine the UTF-8 encodings of the Unicode characters with the following code points:

a) $5a_{16}$.

b) $f5_{16}$.

c) $3f8_{16}$.

d) $8f37_{16}$.

4.2 Determine the UTF-16 encodings of the Unicode characters in exercise 1.

4.3 In this exercise you may need to use the Unicode table which can be found at `www.unicode.org/charts/`.

a) Suppose you save the characters 'æ', 'ø' and 'å' in a file with UTF-8 encoding. How will these characters be displayed if you open the file in an editor using the ISO Latin 1 encoding?

b) What will you see if you do the opposite?

c) Repeat (a) and (b), but use UTF-16 instead of UTF-8.

d) Repeat (a) and (b), but use UTF-16 instead of ISO Latin 1.

Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
32	20	SP	64	40	@	96	60	‘
33	21	!	65	41	A	97	61	a
34	22	"	66	42	B	98	62	b
35	23	#	67	43	C	99	63	c
36	24	$	68	44	D	100	64	d
37	25	%	69	45	E	101	65	e
38	26	&	70	46	F	102	66	f
39	27	’	71	47	G	103	67	g
40	28	(	72	48	H	104	68	h
41	29	)	73	49	I	105	69	i
42	2a	*	74	4a	J	106	6a	j
43	2b	+	75	4b	K	107	6b	k
44	2c	,	76	4c	L	108	6c	l
45	2d	-	77	4d	M	109	6d	m
46	2e	.	78	4e	N	110	6e	n
47	2f	/	79	4f	O	111	6f	o
48	30	0	80	50	P	112	70	p
49	31	1	81	51	Q	113	71	q
50	32	2	82	52	R	114	72	r
51	33	3	83	53	S	115	73	s
52	34	4	84	54	T	116	74	t
53	35	5	85	55	U	117	75	u
54	36	6	86	56	V	118	76	v
55	37	7	87	57	W	119	77	w
56	38	8	88	58	X	120	78	x
57	39	9	89	59	Y	121	79	y
58	3a	:	90	5a	Z	122	7a	z
59	3b	;	91	5b	[	123	7b	{
60	3c	<	92	5c	\	124	7c	|
61	3d	=	93	5d	]	125	7d	}
62	3e	>	94	5e	ˆ	126	7e	~
63	3f	?	95	5f	_	127	7f	BCD

Table 4.2. The ASCII characters with codes 32–127. The character with decimal code 32 is white space, and the one with code 127 is 'delete'.

Dec	Oct	Abbr	CS	Description
0	00	NUL	^@	Null character
1	01	SOH	^A	Start of Header
2	02	STX	^B	Start of Text
3	03	ETX	^C	End of Text
4	04	EOT	^D	End of Transmission
5	05	ENQ	^E	Enquiry
6	06	ACK	^F	Acknowledgment
7	07	BEL	^G	Bell
8	08	BS	^H	Backspace
9	09	HT	^I	Horizontal Tab
10	0a	LF	^J	Line feed
11	0b	VT	^K	Vertical Tab
12	0c	FF	^L	Form feed
13	0d	CR	^M	Carriage return
14	0e	SO	^N	Shift Out
15	0f	SI	^O	Shift In
16	10	DLE	^P	Data Link Escape
17	11	DC1	^Q	XON
18	12	DC2	^R	Device Control 2
19	13	DC3	^S	XOFF
20	14	DC4	^T	Device Control 4
21	15	NAK	^U	Negative Acknowledgement
22	16	SYN	^V	Synchronous Idle
23	17	ETB	^W	End of Trans. Block
24	18	CAN	^X	Cancel
25	19	EM	^Y	End of Medium
26	1a	SUB	^Z	Substitute
27	1b	ESC	^[	Escape
28	1c	FS	^\	File Separator
29	1d	GS	^]	Group Separator
30	1e	RS	^^	Record Separator
31	1f	US	^_	Unit Separator

Table 4.3. The first 32 characters of the ASCII table. The first two columns show the code number in decimal and octal, the third column gives a standard abbreviation for the character and the fourth column gives a printable representation of the character. The last column gives a more verbose description of the character.

Dec	Oct	Char	Dec	Oct	Char
192	c0	À	224	e0	à
193	c1	Á	225	e1	á
194	c2	Â	226	e2	â
195	c3	Ã	227	e3	ã
196	c4	Ä	228	e4	ä
197	c5	Å	229	e5	å
198	c6	Æ	230	e6	æ
199	c7	Ç	231	e7	ç
200	c8	È	232	e8	è
201	c9	É	233	e9	é
202	ca	Ê	234	ea	ê
203	cb	Ë	235	eb	ë
204	cc	Ì	236	ec	ì
205	cd	Í	237	ed	í
206	ce	Î	238	ee	î
207	cf	Ï	239	ef	ï
208	d0	Đ	240	f0	ð
209	d1	Ñ	241	f1	ñ
210	d2	Ò	242	f2	ò
211	d3	Ó	243	f3	ó
212	d4	Ô	244	f4	ô
213	d5	Õ	245	f5	õ
214	d6	Ö	246	f6	ö
215	d7	×	247	f7	÷
216	d8	Ø	248	f8	ø
217	d9	Ù	249	f9	ù
218	da	Ú	250	fa	ú
219	db	Û	251	fb	û
220	dc	Ü	252	fc	ü
221	dd	Ý	253	fd	ý
222	de	Þ	254	fe	þ
223	df	ß	255	ff	ÿ

Table 4.4. The last 64 characters of the ISO Latin1 character set.