Part II: Computer Science Essentials

1. Introduction to computer science

Fundamental notions, systematical overview

What is "Computer Science" / "Informatics" ?

"Computer Science" – science about a tool?

better names would be: "*science of computing*"
or "*data processing science*" (focuses on activity instead of tool)

"Informatics": continental-European for "computer science"

- French: "Informatique" (since 1960s)
- German: "Informatik"

Definition: "Science of the systematical processing of information, especially the automatic processing by use of digital computers".

Latin "informare":
to give structure to something; to educate; to picture

Information:

- independent fundamental entity of the world besides matter and energy
- depends on previous knowledge of the receiver of the information
- various approaches to quantify it
- we can consider information simply as "interpreted data".

Data: represented information
(e.g. text in a book, magnetic patterns on a harddisk, ...)

But:
Hermeneutics – "the art of interpretation" – is *not* part of informatics, despite its name. Social and cultural aspects of information are largely ignored.

"Computer": comes from "to compute"
= "to calculate".

"*Algorithm*":

The word comes from the Persian textbook writer *Abu Ja'far Mohammed ibn Mûsâ al-Khowârizmî*
(= "father of Ja'far Mohammed, son of Moses, coming from Khowârizm" – a town in Usbekistan, today called *Khiva*.)

Al-Khowârizmî lived in Bhagdad, in the "House of Wisdom"
wrote book about calculation:
"Kitab al jabr w'*al-muqabala*" (= "rules of reconstitution and reduction")

– here the word "algebra" comes from!

Modern meaning of "algorithm":

Finite set of rules which specify a sequence of operations in order to solve a certain problem, with the following properties:

1. **Termination**: An algorithm must come to an end after a finite number of steps.

2. **Definitness**: Each step must be defined precisely.

3. **Input**: An algorithm *can* need input values (e.g. numbers).

4. **Output**: An algorithm *must* give one or more output values.

5. **Feasibility**: An algorithm must be feasible; e.g., no intermediate step must depend on the solution of some still unsolved mathematical problem.

(after Knuth 1973)

"*Programme*" (in American English: "program"):

Version of an algorithm which can be read, interpreted and carried out by a computer.

Programming languages were designed to write precise programmes (more precise than possible in our natural language!) suitable for computers.

Some notes concerning the history of *programming:*

Early phases of computer history: *Hardware* (= the machines) was in focus (reason for the name "computer science")
Later: *Software* (= programmes) increasingly important, increasingly expensive in comparison to hardware.

First "programmer": Was a woman (**Lady Ada Lovelace**, daughter of the poet Lord Byron): Developed programs for Babbage's (non-functional) "analytical engine"

An early concept for a programming notation was the "Plankalkül" (Zuse 1944), but it was not used in practice.

Programming these machines: Started with today so-called **"machine languages"** and **"assembler languages"** (both machine-specific).

Later: so-called "high-level languages"
- more abstraction
- better readability for humans
- trying to integrate traditional mathematical notations
- *platform-independent* (not specific to certain machine)

FORTRAN (1954), COBOL (1958), LISP (1960), Pascal (1971), C (1971), C++ (extension of C, 1992), Java (1995), XL (2008) ...

(later more about programming)

Subject areas of computer science

1960s/1970s: Development of specialized university curricula

Basis: Mathematics, electrical engineering; no interest in social or cultural conditions and consequences, or more specifically: in consequences for life at working place and leisure

Classical branches (from first recommendations for curricula in the 1960s): (a) **theoretical** informatics, (b) **technical** informatics, (c) **practical** informatics, (d) **applied** informatics

Theoretical informatics: mathematical basis: not general "theory" (which would include disciplines from the humanities and social sciences relevant to informatics), but specialized "mathematical base". Example questions:

Which problems **can in principle be solved** by a machine?

How can **syntax and semantics** of programming languages be described?

Which kinds of **logic** can be used for automatic problem solving?

How do we measure **how complicated problems are**, for example with respect to time or memory requirements?

Which kinds of problems can be solved with which abstract models of computation?

How can be the **correctness** of a program be **proved** with mathematical exactness?

Technical informatics: focused on **hardware**. Example questions:

How can computational objects and operations be represented with **physical means**?

Which are the **basic parts** from which a computer should be built?

Which are the appropriate **architectural decisions** for a computer?

How can a processor be organized in order to execute a special kind of program especially quickly?

How is information **stored** for quick access with small cost?

Which are the technical conditions for building **networks** from separate computers?

How do we build computers which **survive some defects**?

Practical informatics: **non application specific programming**. Example questions:

Which are the standard problems occurring **in many application areas**, and how can they be solved?

Which **data structures** allow efficient solving of problems, and which algorithms are best used on these data structures?

What types of **programming languages** are best suited to different types of problems?

How must **service programs** be organized which provide the user with an easier to use view of the machine than the bare hardware would do?

How are high-level programs **translated** into a form which can be executed by the underlying hardware?

How does one design **user interfaces** for end users?

How does one organize the **development process** of large software systems? ("Software engineering")

Applied informatics: programming for **specific application fields**. Example questions:

How are **graphical objects** represented in the computer, and how can the be visualized?

Which **numerical methods** exist to model states and processes happening in natural environments?

How should **data base systems** be structured to support the work processes in a company?

Which techniques exist to simulate the working of the **human mind** with computers?

What consequences has the use of computers for the **quality of life**, both in general and at the working place in particular?

Informatics in the social context:

What **ethical questions** arise from the use of computers, and how can they be answered?
(data security, privacy questions, computer viruses, hackers, violence-promoting games, software piracy, ownership of software and ideas, the open-source idea, use of information technology for warfare, for crime, for sexual exploitation, for terrorism...)

How does the use of computers influence our **way of thinking** (about the world, about humans, about the mind, about personal relationships of people...)?

How can computers, the Web and the "Web 2.0" (Facebook, Twitter, Wikipedia etc.) be used to improve education / autonomy of people / human rights / political participation... ?

What are possible dangers / cases of misuse?

2. Representation and measurement of information

In digital computers and media, all data are represented by combinations of only 2 elementary states: 0 and 1
(can be "charged" / "not charged", "on" / "off", "magnetized" / "not magnetized", "open" / "closed", "high current" / "low current", "plus" / "minus" etc.)

The smallest amount of information is thus the *bit* (binary digit). It expresses which of two alternatives is the case. The alternatives are often written
0 and 1, or (sometimes) 0 and L.

n bits: represent one out of 2^n alternatives.

Codes

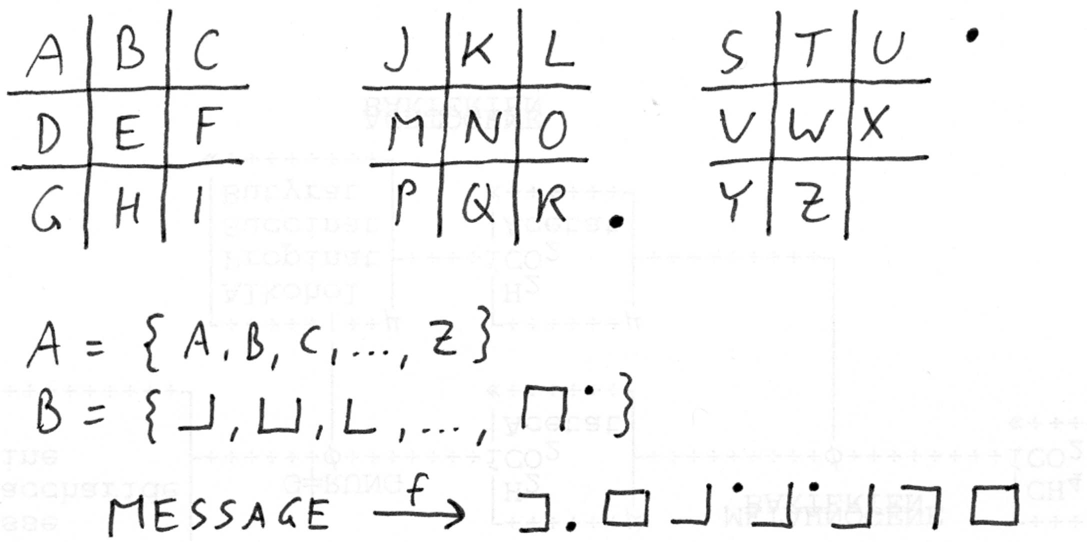
To represent information in a computer, we must *encode* all with the two symbols 0 and 1 !

What is a *code* ?

Code (1): A mapping f: $A \to B$ from a set A of elements to be stored or transferred to a set B used for storage or transfer.

Code (2): The set B from definition (1).

Example:



digital (discrete) and **analogue** (continuous) codes

Analogue computers (representation of quantities with continuously changing quantities): have vanished

Example: Vinyl records (analogue) vs. compact disks (discrete)

Benefit of discrete data representations: avoiding **noise**

For digital computers, we need *binary* codes:
B is a set of combinations of 0 and 1.

Examples:

For the primary **compass direction**: two bits necessary, and some convention which bit-pair represents which direction. Example code:

$$\{N, E, S, W\} \to \{0,1\}^2, N \mapsto 00, E \mapsto 01, S \mapsto 10, W \mapsto 11$$

For **Boolean** values 'True' and 'False':

$$\{T, F\} \to \{0,1\}, T \mapsto 1, F \mapsto 0$$

For **numbers** 0 to 9: Binary Coded Decimal (BCD, non-total code, i.e. some combinations are unused)

$$\begin{gathered}\{0,1,\ldots,9\} \to \{0,1\}^4 \\ 0 \mapsto 0000, 1 \mapsto 0001, 2 \mapsto 0010, 3 \mapsto 0011, \\ 4 \mapsto 0100, 5 \mapsto 0101, 6 \mapsto 0110, 7 \mapsto 0111, \\ 8 \mapsto 1000, 9 \mapsto 1001\end{gathered}$$

Multiples of bits

Bits seldom occur as singles. Certain multiples of bits are used as *units for information (storage) capacity*.

1 Byte: 8 bits (can represent 1 of 2^8 = 256 alternatives).

Example: one of the integer numbers between –128 and +127.

1 Halfbyte: 4 bits.

Typically, memory stores are built for *multiples of bytes*.

Prefixes: kilo, mega, giga, tera, peta, exa

- used in physics for the factors 10^3, 10^6, 10^9, 10^{12}, 10^{15},10^{18}
- in computer science often used for the factors 2^{10}, 2^{20}, 2^{30}, 2^{40}, 2^{50}, 2^{60}, which are slightly larger

abbre-viation	meaning	factor
KB	Kilobytes	2^{10} = 1024
MB	Megabytes	2^{20} = 1,048,576
GB	Gigabytes	2^{30} = 1,073,741,824
TB	Terabytes	2^{40} = 1,099,511,627,776
PB	Petabytes	2^{50} = 1,125,899,906,842,624
EB	Exabytes	2^{60} = 1,152,921,504,606,846,976

Representation of numbers in the computer

Number systems

Question: How to represent numbers?
We focus on *positive integers* here.

Decimal number system: base 10; each digit represents a multiple of an exponent of 10. Digits 0..9.

Example: $123.456_{10} = 1*10^2 + 2*10^1 + 3*10^0 + 4*10^{-1} + 5*10^{-2} + 6*10^{-3}$.

Binary number system: base 2. Only two digits: 0 and 1.

Example: $1101.01_2 = 1*2^3 + 1*2^2 + 0*2^1 + 1*2^0 + 0*2^{-1} + 1*2^{-2} = 13.25_{10}$.

Hexadecimal system (better but unhistorical name: sedecimal number system): Base 16, digits 0..9,A..F. One digit for four bits. Examples: $A2.8_{16} = 162.5_{10}, FF_{16} = 255_{10}$.

The additional digits in the hexadecimal system:
A = 10, *B* = 11, *C* = 12, *D* = 13, *E* = 14, *F* = 15.

Transformation from one number system to the other:

- Special case (easy): from binary to hexadecimal

Every 4 binary digits correspond directly to a hexadecimal digit

Example: <u>0000</u> <u>0010</u> <u>1100</u> <u>0110</u>
$\rightarrow$ 0 2 C 6

- from arbitrary system to decimal:
  Horner scheme

Input: $z_{n-1}\ z_{n-2}\ \ldots\ z_0$ to base b

start with $h_{n-1} = z_{n-1}$

calculate for $k = n-1, n-2, \ldots, 1$:

$$h_{k-1} = h_k * b + z_{k-1}$$

Output: $z = h_0$

Example:

Input: binary number 1010 ($n = 4$, $b = 2$)

Start: $h_{n-1} = h_3 = z_3 = 1$

$k = n-1 = 3$: $h_2 = h_3 * 2 + z_2 = 1*2 + 0 = 2$

$k = 2$: $h_1 = h_2 * 2 + z_1 = 2*2 + 1 = 5$

$k = 1$: $h_0 = h_1 * 2 + z_0 = 2*5 + 0 = \mathbf{10} = z$

- from decimal to arbitrary:
  Inverse Horner scheme

start with $h_0 = z$ (= input)

calculate for $k = 1, 2, 3, \ldots$:

$$z_{k-1} = h_{k-1} \text{ mod } b,$$
$$h_k = h_{k-1} \text{ div } b$$

(mod: rest when dividing by *b*, div: integral part from dividing by *b*)

Output: $z_{n-1}\, z_{n-2} \ldots z_0$ to base b

Example:

Input: decimal number 34, transform in ternary system ($b = 3$)

Start: $h_0 = 34$

$k = 1$: $z_0 = h_0 \text{ mod } 3 = 34 \text{ mod } 3 = 1$,
$h_1 = h_0 \text{ div } 3 = 34 \text{ div } 3 = 11$

$k = 2$: $z_1 = h_1 \text{ mod } 3 = 11 \text{ mod } 3 = 2$,
$h_2 = h_1 \text{ div } 3 = 11 \text{ div } 3 = 3$

$k = 3$: $z_2 = h_2 \text{ mod } 3 = 3 \text{ mod } 3 = 0$,
$h_3 = h_2 \text{ div } 3 = 3 \text{ div } 3 = 1$,

$k = 4$: $z_3 = h_3 \text{ mod } 3 = 1 \text{ mod } 3 = 1$,
$h_4 = h_3 \text{ div } 3 = 1 \text{ div } 3 = 0$ (Stop)

$\Rightarrow z = 1021$

Remark:

Arbitrary real numbers can also be represented using an arbitrary integer $b > 1$ as base.
Digits after the dot are interpreted as coefficients of b^{-n} ($n = 1, 2, 3, ...$).

Example:
0.111_2 (base b=2)

$= 1/2 + 1/4 + 1/8 = 7/8 = 0.875_{10}$

Representation of numbers in the computer

For positive integers, basically the *binary number system* is used.

But: Numbers are usually stored in sections of memory of fixed size (for reasons of organization of memory access in the computer).
Integer representation in finite cells ("words" with fixed length):

Computer memory: organized in **finite cells**. Typically: Multiples of a byte.

How to store numbers in a 4-byte cell? Some encoding necessary. 2^{32} different values can be represented.

Example: $0 \ldots 2^{32} - 1$ can be represented as binary numbers.

Example including negative numbers: $-2^{31} \dots 2^{31}-1$ can be represented as two's complements numbers.

Two's complement: Most used representation for integers from range $-2^{n-1} \dots 2^{n-1}-1$ (with n-bit cell).

Non-negative numbers: Are represented simply as binary numbers. Using n bits, the highest bit is always 0.

Negative numbers: (a) Represent their absolute value as binary number, (b) then invert all bits (including the infinite number of leading zeros, resulting in an infinite number of leading ones), and (c) add a 1. The last n bits are the two's complement of the value to be represented.

Example for the "Two's complement":

8-bit two's complement representation of –77

1. Represent +77 as a binary number: 1001101
2. Invert all bits, including the leading 0s: ...1110110010
3. Add 1: ...1110110011
4. Use only the lowest (= rightmost) 8 bits: 10110011

Notice:
For 16-bit cells, the result would be 1111111110110011.

decimal system	8-bit two's complement
-128	1000 0000
-127	1000 0001
-126	1000 0010
...	...
-2	1111 1110
-1	1111 1111
0	0000 0000
1	0000 0001
...	...
126	0111 1110
127	0111 1111

Properties of the two's complement:

Code represents numbers $-2^{n-1} \dots 2^{n-1} - 1$.

High bit represents **sign**.

Minimal value represented by $1000\ldots$, maximal by $0111\ldots$.

-1 represented by $111\ldots$.

Floating-point representations

Built analogously to the "scientific representation" of numbers in the form *m* * 10^e

- but using the binary system:

Represent numbers in the form

$$s * m * 2^e$$

with sign s ($+1$ or -1), non-negative mantissa m, and integer exponent e.

Representation is **normalized** if $1 \leq m < 2$.

Finite number of bits for sign, mantissa and exponent; often used: 32 bits (single precision), 64 bits (double precision), 80 bits (extended precision)

Typical layout of 32-bit floating point number:

Bit 31: represents s (1: negative; 0: positive)

Bits 30..23 (8 bits): represent e: Binary representation of $e + 127$, which allows the values $-126 \ldots 127$. Value 0 is used in representation of number 0 and of unnormalized numbers. Value 255_{10} used to represent infinity and other exceptional values.

Bits 22..0 (23 bits): represent m, by binary representation of the integer part of $m * 2^{23}$, without the leading 1.

Example: representing $+26.625$ as a 32-bit normalized floating point number: $26.625_{10} = 11010.101_2$. Normalizing yields $1.1010'1010_2 * 2^4$. 32-bit floating point number (s=0, e=131_{10}):

$$0'10000011'10101010000000000000000$$

Digital representation of text

based on representation of *letters*
- depending on the alphabet: certain number of bits necessary
- for 26 letters: at least 5 bits necessary ($2^4 = 16 < 26$, $2^5 = 32 > 26$)
- but also encoding of digits, special signs, upper- and lower-case letters... desirable

traditional 7-bit code:
ASCII (= *American Standard Code for Information Interchange*)
ISO-646 norm
later extended to 8-bit code

examples: 00110000 = hex 30 = 48_{10} = digit 0
00110001 = hex 31 = 49_{10} = digit 1
...
00111010 = hex 3A = 58_{10} = ':'
...
01000001 = hex 41 = 65_{10} = 'A'
01000010 = hex 42 = 66_{10} = 'B'
...
01100001 = hex 61 = 97_{10} = 'a'
...

ASCII Table:

Non-printable characters

Dez	Okt	Hex	Char	Code	Remark
0	000	0x00	Ctrl-@	NUL	Null prompt
1	001	0x01	Ctrl-A	SOH	Start of heading
2	002	0x02	Ctrl-B	STX	Start of text
3	003	0x03	Ctrl-C	ETX	End of Text
4	004	0x04	Ctrl-D	EOT	End of transmission
5	005	0x05	Ctrl-E	ENQ	Enquiry
6	006	0x06	Ctrl-F	ACK	Acknowledge
7	007	0x07	Ctrl-G	BEL	Bell
8	010	0x08	Ctrl-H	BS	Backspace
9	011	0x09	Ctrl-I	HT	Horizontal tab
10	012	0x0A	Ctrl-J	LF	Line feed
11	013	0x0B	Ctrl-K	VT	Vertical tab
12	014	0x0C	Ctrl-L	FF	Form feed
				NP	New page
13	015	0x0D	Ctrl-M	CR	Carriage return
14	016	0x0E	Ctrl-N	SO	Shift out
15	017	0x0F	Ctrl-O	SI	Shift in
16	020	0x10	Ctrl-P	DLE	Data link escape
17	021	0x11	Ctrl-Q	DC1	X-ON
18	022	0x12	Ctrl-R	DC2	
19	023	0x13	Ctrl-S	DC3	X-Off
20	024	0x14	Ctrl-T	DC4	
21	025	0x15	Ctrl-U	NAK	No achnowledge
22	026	0x16	Ctrl-V	SYN	Synchronous idle
23	027	0x17	Ctrl-W	ETB	End transmission blocks
24	030	0x18	Ctrl-X	CAN	Cancel
25	031	0x19	Ctrl-Y	EM	End of medium
26	032	0x1A	Ctrl-Z	SUB	Substitute
27	033	0x1B	Ctrl-[	ESC	Escape
28	034	0x1C	Ctrl-\	FS	File separator
29	035	0x1D	Ctrl-]	GS	Group separator
30	036	0x1E	Ctrl-^	RS	Record separator
31	027	0x1F	Ctrl-_	US	Unit separator
127	0177	0x7F		DEL	Delete or rubout

Printable characters

Dez	Okt	Hex	Char	Remark
32	040	0x20		blank
33	041	0x21	!	exclamation mark
34	042	0x22	"	quotation mark
35	043	0x23	#	
36	044	0x24	$	Dollar character
37	045	0x25	%	
38	046	0x26	&	
39	047	0x27	'	apostroph
40	050	0x28	(	
41	051	0x29	)	
42	052	0x2A	*	asterisk
43	053	0x2B	+	plus sign
44	054	0x2C	,	comma
45	055	0x2D	-	minus sign
46	056	0x2E	.	dot
47	057	0x2F	/	slash
48	060	0x30	0	
49	061	0x31	1	
50	062	0x32	2	
51	063	0x33	3	
52	064	0x34	4	
53	065	0x35	5	
54	066	0x36	6	
55	067	0x37	7	
56	070	0x38	8	
57	071	0x39	9	
58	072	0x3A	:	colon
59	073	0x3B	;	semicolon
60	074	0x3C	<	less than
61	075	0x3D	=	euqality character
62	076	0x3E	>	greater than
63	077	0x3F	?	interrogation mark
64	0100	0x40	@	at
65	0101	0x41	A	
66	0102	0x42	B	
67	0103	0x43	C	
68	0104	0x44	D	
69	0105	0x45	E	
70	0106	0x46	F	
71	0107	0x47	G	
72	0110	0x48	H	
73	0111	0x49	I	
74	0112	0x4A	J	
75	0113	0x4B	K	
76	0114	0x4C	L	
77	0115	0x4D	M	
78	0116	0x4E	N	
79	0117	0x4F	O	

80	0120	0x50	P	
81	0121	0x51	Q	
82	0122	0x52	R	
83	0123	0x53	S	
84	0124	0x54	T	
85	0125	0x55	U	
86	0126	0x56	V	
87	0127	0x57	W	
88	0130	0x58	X	
89	0131	0x59	Y	
90	0132	0x5A	Z	
91	0133	0x5B	[	
92	0134	0x5C	\	backslash
93	0135	0x5D	]	
94	0136	0x5E	^	caret
95	0137	0x5F	_	low line
96	0140	0x60	`	back quote
97	0141	0x61	a	
98	0142	0x62	b	
99	0143	0x63	c	
100	0144	0x64	d	
101	0145	0x65	e	
102	0146	0x66	f	
103	0147	0x67	g	
104	0150	0x68	h	
105	0151	0x69	i	
106	0152	0x6A	j	
107	0153	0x6B	k	
108	0154	0x6C	l	
109	0155	0x6D	m	
110	0156	0x6E	n	
111	0157	0x6F	o	
112	0160	0x70	p	
113	0161	0x71	q	
114	0162	0x72	r	
115	0163	0x73	s	
116	0164	0x74	t	
117	0165	0x75	u	
118	0166	0x76	v	
119	0167	0x77	w	
120	0170	0x78	x	
121	0171	0x79	y	
122	0172	0x7A	z	
123	0173	0x7B	{	
124	0174	0x7C	|	
125	0175	0x7D	}	
126	0176	0x7E	~	

ASCII not sufficient for alphabets of the non-Anglo-american world (not even for European alphabets with ä, ö, ü, ß, é, ø, ñ, å...)

Unicode:
2 byte (= 16 bit) code for multilingual text processing
- can represent 65536 characters

amongst them: 27786 Chinese-Japanese-Korean characters
11172 Hangul characters (Korean)
ancient Nordic runes
Tibetan characters
Cherokee characters ...

complete list see **http://www.unicode.org/charts/**

Unicode "Escape sequence" (to utilise it in the programming language Java):
e.g., \u0041 = 'A' (0041 = hexadecimal representation)

Some characters occur more frequently in texts than others:
better use *variable-length* code

UTF-8: Universal Transformation Format
Characters encoded with variable number of bytes
$\Rightarrow$ for texts with many ASCII characters (like on many web pages) shorter as Unicode

Strings (or *words*): sequences of characters
encoded by sequences of the corresponding code words

Digital representation of pictures

Gray levels: encode each gray level by a number from a fixed interval (e.g. 0, 1, ..., 255: 8-bit representation – 0 = black, 255 = white)

Colours:
several colour models possible
the most frequently used one: *RGB model*
(red / green / blue: primary colours for additive colour composition)
Each colour from a certain range ("gamut") can be mixed from these primary colours

examples with 8-bit intensities:

black	(0, 0, 0)
white	(255, 255, 255)
medium gray	(127, 127, 127)
red	(255, 0, 0)
green	(0, 255, 0)
blue	(0, 0, 255)
light blue	(127, 127, 255)
yellow	(255, 255, 0)

Pictures:
typically represented as raster images – rectangular array (matrix) of *pixels*, each pixel represented by its 3 colour values.

Representation of text documents (book pages, web pages...)

Level of representation is important.

(1) Is there text on the page? – One bit.

(2) What is the text on the page? –
Representation of letter sequence (e.g., string of ASCII characters).

(3) What is the exact layout of the text on the page? –
"formatted text"
- use special characters for formatting, or
- represent the page by a rasterized black-and-white image.

Text documents with graphical elements:

- represent all as a single raster image, or

- use combined representation: several data files, one for the text, the other for the pictorial parts
$\rightarrow$ HTML web pages are built like this

file <name>.html or <name>.htm contains text, layout information and links to other pages
files <name>.gif or <name>.jpg or <name>.png contain images

Messages and redundancy

Message: A finite sequence of letters, used to transfer some information via encoding/transfer/decoding

Signal: The physical representation of the message (examples: as voltage pattern or light pattern)

Redundancy: Part of a message which is not necessary for the transferred information (later explained more exactly)

Error correction by **redundant** codes: Natural languages allow to detect many errors.

Example in informatics: **Parity bits**. Even parity: 9 bits per byte. 9th bit makes number of one-bits even. Allows detection of single-bit errors. Computer memory sometimes uses 9 bits per byte for this purpose.

Other example: ISBN code (International Standard Book Number) – last character is a parity character

Entropy and quantification of information

Shannon's information theory:
Information as a measurable, statistical property of signals

How can we measure information and redundancy of characters in a message?

Assumption: N-character alphabet $\{ x_1, x_2, ..., x_N \}$

Number of bits per character:

$$H_0 = \log_2 N$$

(Remember: $\log_2 N = (\log N)/(\log 2)$)

Information content of a single character x_i : $\log_2 \frac{1}{p(x_i)}$

Here, $p(x_i)$ is the probability of x_i .

Entropy = *average value of information content* of all characters

$$= H = \sum_{k=1}^{N} p(x_k) * \log_2 \frac{1}{p(x_k)}$$

Binary encoding needs at least, on average, H bits per character.

Redundancy: $R = H_0 - H$.

Example: Four-letter alphabet $\{a, b, c, d\}$

Probabilities: $p_a = 0.5, p_b = 0.25, p_c = 0.125, p_d = 0.125$

Thus:

$H_0 = 2$ bits per character encodable

Entropy: $0.5 * 1 + 0.25 * 2 + 0.125 * 3 + 0.125 * 3 = 1.75$ bits per character encoded

Redundancy: 0.25 bits per character

Examples:

– $a \mapsto 00, b \mapsto 01, c \mapsto 10, d \mapsto 11$: on average 2 bits per character

– $a \mapsto 0, b \mapsto 10, c \mapsto 110, d \mapsto 111$: on average 1.75 bits per character (optimal, no redundancy)