Number Base Case
$N = d_n R^n + d_1 R^1 + d_0 R^0$
the d specifies the Number system -> d_2 == binary
can also be written as R_2
This can also be used to expand numbers:
$N_{10} 255 = 2 * 10^2 + 5 * 10^1 + 5 * 10^0$
$N_2 110 = 1 * 2^2 + 1 * 2^1 + 0 * 2^0 => N_{10} 6$

Quantities:
N -> natural numbers | | **Z** -> full numbers
Q -> rational numbers | | **R** -> real numbers

Common number systems:
Decimal: $N_{10} = n * 10^n .. 0 * 10^0$
Binary: $N_2 = n * 2^n .. 0 * 2^0$
$2^{10} = 1024, 2^9 = 512, 2^8 = 256, 2^7 = 128, 2^6 = 64,$
$2^5 = 32, 2^4 = 16, 2^3 = 8, 2^2 = 4, 2^1 = 2, 2^0 = 1$
Hexadecimal: $N_1 6 = n * n^{16} .. 0 * 16^0$
notation: 0 1 2 3 4 5 6 7 8 9 A B C D E F
$16^5 = 1048576, 16^4 = 65536, 16^3 = 4096, 16^2 = 256,$
$16^1 = 16, 16^0 = 1$

Modulo
8 mod 4 = (8) -> 0 , 8 mod 3 = (6) -> 2 , 8 mod 5 = (5) -> 3
if x<y in x mod y then the result will always be x!
any negative numbers can be considered as NOTnegative
aka only absolute values! modulo deals with |x|
many programming languages actually do not follow this!
they have their own implementation of modulo.
$5 \equiv 3$ mod 2 -> as 5 mod 2 = 1 and 3 mod 2 = 1

Codeword length
Byte = 8 bit || Word = 16 or 32 bit
TCP packet = 1024 bit

Cyclic group

Es sei $F(a) = a^3 + a + 1 = 0$,

- Dann können wir zunächst festhalten
- a = a
- a² = a² aber
- a³ = ***a*+1**
- a⁴ = a(***a* + 1**) = a² + a
- a⁵ = a(a² + a) = a³ + a² = a² + ***a*+1**
- a⁶ = a(a² + ***a* +1**) = a³ + ***a*² +*a* = *a*+1 + *a*² + *a* = *a*² + 1**
- a⁷ = a(***a*² + 1**) = a³ + a = ***a*+1**+ a = 1
- a⁸ = a : der Zyklus beginnt von vorne!
- **{0, 1, *a*, *a*², *a*+1, *a*² + a, *a*² + *a*+1, *a*² + 1 }**
- ***{000, 001, 010, 100, 011, 110, 111, 101}***

WHAT THE FUCK

Result Quantity the result of all possible outcomes
it is denoted with: Ω
A single element of the result list is: ω -> $\omega \in \Omega$
The list of results is $|\Omega|$
Example Dice roll: $\Omega = \{1, 2, 3, 4, 5, 6\}$

Probability: $P(A) = \frac{\text{best results}}{\text{all results}} = \frac{|A|}{|\Omega|} = \frac{|A|}{n}$

So what is the probability of rolling a 6?
$P(\text{desired number to roll}) = \frac{\text{only 1 good result!}}{\text{6 possible results}} = \frac{1}{6}$
hence the chance is 1 in 6
Why this complicated method? You can modify desired results!
just change the A in P(A)!

Inverse Probability: P(inverse) = 1 - P(A)
dice -> $1 - \frac{1}{6} = \frac{5}{6}$

Addition rule:
$$P(A \cup B) = P(A) + P(B) - P(A \cap B)$$
!!The last part is needed, as otherwise the number
would exceed the possible states!!
$P(A \cup B \cup C) = P(A) + P(B) + P(C) - P(A \cap B) - P(A \cap C) - P(B \cap C) + P(A \cap B \cap C)$

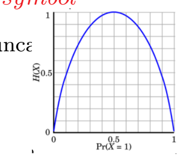
Amount of possibilities:
ordered probes with replication:
2 coins, head and tail, possibilities? **k**=head/tail=2 **n**=coins=2
$$\Omega = n^k = 2^2$$
ordered probes without replication:
5 dices. How many combinations?
dice numbers = **n** = 6 (1-6), dice amount = **k** = 5
possibilities = $\Omega = \frac{n!}{(n-k)!} = \Omega = \frac{6!}{(6-5)!} = 720$
Or this:
$$\Omega = \Pi_n^{n-k+1} n = \Pi_6^{6-5+1} 6 = 2 * 3 ... 5 * 6 = 720$$
unordered probes wihout replication:
25 players, each should only play once with the other.
$\Omega = \frac{n!}{k!(n-k)!} -> \frac{25!}{2!(25-2)!} -> \frac{\text{too big}}{\text{too big}} = 300$
as you can see the bottom is a BIG calculation, so
$$\Omega = \frac{\Pi_n^{n-k+1} n}{k!} -> \frac{\Pi_{25}^{25-2+1} 25}{2!} -> \frac{24 * 25}{2} = 300$$
Note that **k** can also be defined as the
length of the tuple we want to receive.
-> (Player,Player) - > 2

Source to Sink Information

Nachricht (Darstellung & Bedeutung)	redundant	nicht-redundant
irrelevant	Zeichenvorrat bei Quelle und Senke verschieden	
relevant	vorhersagbar	Information

Entropy
information content
this essentially just us how many bits are needed
k is base state count -> bit = 2
and **N** is the full number of states
example: list True,False,True,False 4 states total, base 2.
$$H_0 = log_k(N)[k] -> H_0 = log_2(4)[bit] = 2$$

information flow
essentially information content over time
$$H_0^* = \frac{log_2(N)}{\tau}[\frac{bit}{s}]$$
information quantity / Surprise
$$I(x_k) = -log_2(P(x_k))[bit]$$
Entropy (Surprise per element)
0 means no symbols. 1 means perfect balance 50-50
$$H(X) = \Sigma_{k=1}^N P(x_k) * I(x_k)[\frac{bit}{symbol}]$$
where X is the list of symbols



Sink Redundance / Code Redunca
$$R_Q = H_0 - H(X)[\frac{bit}{symbol}]$$
$$R_c = L - H(X)[\frac{bit}{symbol}]$$
Code Word Length
$$L(x_k) = \text{rounded}(I(x_k))[bit]$$
Median Code Word Length
$$L = \Sigma_{k=1}^N P(x_k) * L(x_k)[\frac{bit}{symbol}]$$
Entropy of the entire Code
$$H_c(X) = \Sigma_{k=1}^N P(x_k) * L(x_k)[\frac{bit}{symbol}]$$
H_c can be a real number -> $H_c \in \mathbb{R}$

Für jede beliebige zugehörige Binärcodierung mit Präfixeigenschaft ist die mittlere Codewortlänge nicht kleiner als die Entropie $H(X)$:	Für jede beliebige Quelle kann eine Binärcodierung gefunden werden, so dass die folgende Ungleichung gilt:
$H(X) \leq L$	$H(X) \leq L \leq H(X) + 1$

Sink without memory
$$P(x_k, y_k) = P(x_k) + P(y_i)$$
Sink with memory
$$P(x_k, y_i) = P(x_k) + P(x_k|y_i)$$
Entropy without memory / Combined Entropy
$$H(H, Y) = \Sigma_{x_k}^N \Sigma_{y_i}^N P(x_k, y_i) * (-log_2(P(x_k, y_i))$$
or: $H(X, Y) = H(X) + H(Y)$
Entropy with memory
$$H(H, Y) = \Sigma_{x_k}^N \Sigma_{y_i}^N P(x_k) * P(x_k, y_i) * (-log_2(P(x_k) * P(x_k|y_i))$$

Encoding of Symbols
- **Ordne die Zeichen gemäss ihrer Auftrittswahrscheinlichkeit**
- **Die beiden Zeichen mit der kleinsten Auftrittswahrscheinlichkeit haben die gleiche CW-Länge L_N**
- **Sei L_N die mittlere CW-Länge für eine Quelle mit N Zeichen und L_{N-1} die mittlere CW-Länge für den Fall, dass die beiden letzten zu einem einzigen Zeichen zusammengefasst werden, dann gilt:**

$$L_N - (p(x_{N-1}) + p(x_N)) \cdot L(x_N) = L_{N-1} - (p(x_{N-1}) + p(x_N)) \cdot (L(X_N) - 1)$$
$$\Rightarrow L_N = L_{N-1} + p(x_{N-1}) + p(x_N)$$

1	2	3	4	5	6	7	8	9
0.22	0.19	0.15	0.12	0.08	0.07	0.07	0.06	0.04

1	2	3	4	8	9	5	6	7
				0	1			
0.22	0.19	0.15	0.12	0.1		0.08	0.07	0.07

1	2	3	6	7	4	8	9	5
			0	1		0	1	
0.22	0.19	0.15	0.14		0.12	0.1		0.08

1	2	8	9	5	3	6	7	4
		00	01	1		0	1	
0.22	0.19	0.18			0.15	0.14		0.12

continue this pattern until every symbol has a code
note the extra 0 on every step

Run Length Encoding RLE/RLC
- Quelltext w: Agggbbehfffgggg => |w|=15
- Codiert w_c : A3g2beh3f4g => | w_c |= 11

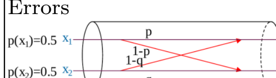
A + 3 x g + 2 x b + e + h + 3 x f + 4 x g

shortening of length
by compressing repetition.

Encoder and Decoder
You need to either choose 1 or 0 as the starting
bit. After that the decoder can print out the correct code.

Chiffre text
You can "encrypt" your data by
shifting the codes by a certain amount.
In the caesar chiffre this is done with the number 4. a -> e
Please do not use this, use RSA or other algorithms.

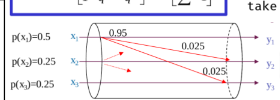
Errors



$p(y_1) = p(x_1) \cdot p + p(x_2) \cdot (1-q)$
$p(y_2) = p(x_1) \cdot (1-p) + p(x_2) \cdot (q)$

$p(Y|X) = \begin{bmatrix} p & 1-p \\ 1-q & q \end{bmatrix} \mapsto \begin{bmatrix} \sum = 1 \\ \sum = 1 \end{bmatrix}$

1-p and 1-q are the chance for
error. Which we of course have to
take into account.

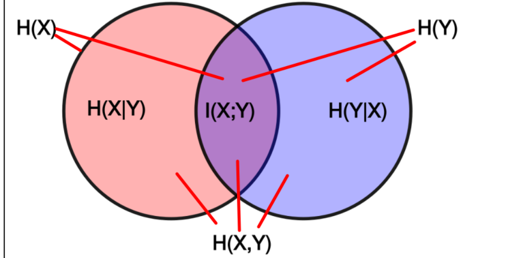


$p(Y|X) = \begin{bmatrix} 0.95 & 0.025 & 0.025 \\ 0.025 & 0.95 & 0.025 \\ 0.025 & 0.025 & 0.95 \end{bmatrix}$

$\begin{pmatrix} p(y_1) \\ p(y_2) \\ p(y_3) \end{pmatrix} = \begin{pmatrix} p(x_1) \cdot p(y_1|x_1) + p(x_2) \cdot p(y_1|x_2) + p(x_3) \cdot p(y_1|x_3) \\ p(x_1) \cdot p(y_2|x_1) + p(x_2) \cdot p(y_2|x_2) + p(x_3) \cdot p(y_2|x_3) \\ p(x_1) \cdot p(y_3|x_1) + p(x_2) \cdot p(y_3|x_2) + p(x_3) \cdot p(y_3|x_3) \end{pmatrix}$

$\begin{bmatrix} 0.4875 \\ 0.25625 \\ 0.25625 \end{bmatrix} = \begin{bmatrix} 0.5 \cdot 0.95 + 0.25 \cdot 0.025 + 0.25 \cdot 0.025 \\ 0.5 \cdot 0.025 + 0.25 \cdot 0.95 + 0.25 \cdot 0.025 \\ 0.5 \cdot 0.025 + 0.25 \cdot 0.025 + 0.25 \cdot 0.95 \end{bmatrix}$

Conditional Entropy -> Entropy of Y given X
$$H(Y|X) = \Sigma_{k=1}^N \Sigma_{i=1}^N P(x_k, y_i) * (-log_2(\frac{P(x_k, y_i)}{P(x_k)}))$$
Chain Rule
$$H(Y|X) = H(X, Y) - H(X) \ || \ H(Y \backslash X)$$
Bayes Rule
$$H(Y|X) = H(X|Y) - H(X) + H(Y) \ || \ H(Y \backslash X)$$



Transinformation
likelyhood of information being correct at arrival.
$$T = H(X) - H(X|Y) \ || \ H(Y) - H(Y|X)$$
or: $|(X; Y)$

Hamming distance / distance to next valid codeword
$$h = Min_{i,j}(d(x_i, x_j)) -> h = \text{message points} + 1$$
error detection distance
the amount of bits that differ from input to output
$$e^* = h - 1$$
error correction distance for h even
$$h = 2e + 2 -> e = \frac{h-2}{2}$$
error correction distance for h uneven
$$h = 2e + 1 -> e = \frac{h-1}{2}$$
Consider the valid input either 111 or 000.
The Hamming distance **h** is therefore 3 bits.
The detection distance e^* is 3 - 1
Due to h being uneven, the correction distance **e** is $\frac{h-1}{2}$
which results in 1.

tighly packed coderoom
n = dimension of code
m = dimension of messages $2^m * \Sigma_{w=0}^e \binom{n}{w} \leq 2^n$
k = dimension of control -> n = m + k
The code is considered to be tightly packed
if the equation has the result 2. aka == not smaller.

m=2 | k=1

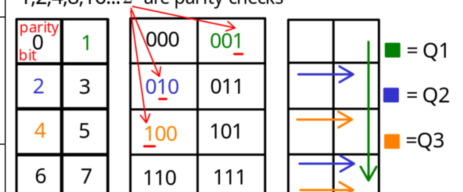
x_1	x_2	x_3
0	0	0
0	1	1
1	0	1
1	1	0
0	0	1
0	1	0
1	0	0
1	1	1

```
0 = (0 + 0) mod 2 = 0 OK
1 = (0 + 1) mod 2 = 1 OK
1 = (1 + 0) mod 2 = 1 OK
0 = (1 + 1) mod 2 = 0 OK
```

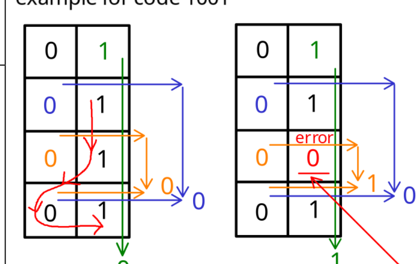
$x_3 = (x_1 + x_2) \bmod 2$

```
1 = (0 + 0) mod 2 = 0 NOT OK
0 = (0 + 1) mod 2 = 1 NOT OK
0 = (1 + 0) mod 2 = 1 NOT OK
1 = (1 + 1) mod 2 = 0 NOT OK
```

Hamming Codes
The hamming code is very easy to implement
$$\Sigma_i x_i * \vec{P_i} \equiv \vec{0} \, mod 2$$
The syndrome $\vec{Z} = \Sigma_i x_i * \vec{P_i} mod 2$
1,2,4,8,16...2^x are parity checks



example for code 1001



$\vec{Z}$=000=no error $\vec{Z}$=101=error at 101 = 5

note that the 001 010 100 of the parity checks are
simply the unit vector $\vec{0}$!!!

parity checks needed:

$par = log_2$(bit amount of code)

1101 = 4 bits -> 3 parity checks
as 4 can be displayed by 3 bits -> 100

OR we can use XOR

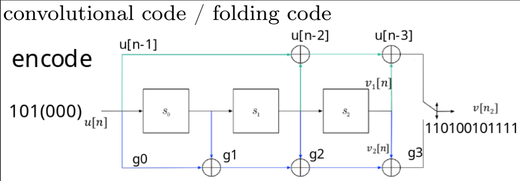
write down all the positions with 1 in binary:

001		001
011		011
101	error ->	
⊕111		⊕111
000		101

errors can be represented by either removing one bit from the calculation -> setting it to 0 or adding it, setting it to 1

convolutional code / folding code

encode



$u[n]$	s_0	s_1	s_2	$v_1[n]$	$v_2[n]$	$v[n_2]$
-	0	0	0	-	-	-
1	0	0	0	1	1	11
0	1	0	0	0	1	01
1	0	1	0	0	0	00
0	1	0	1	1	0	10
0	0	1	0	1	1	11
0	0	0	1	1	1	11
-	0	0	0	-	-	-

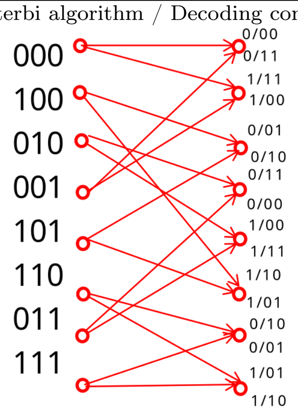
this algorithm is heavily dependent on the configuration, aka the more s states you have the more bits you get as the output.

$\{u[n]\} = \{u_1, u_2, \dots, u_n\}$

$\{v[n]\} = \{[g^0 u_{[n]} + g^1 u[n-1] + g^2 u[n-2] + g^3 u[n-3]], g^0 u[n+1-m], \dots\}$

$\{v[n]\} = \sum_{m=0}^{M} g^m u_{[n-m]}$

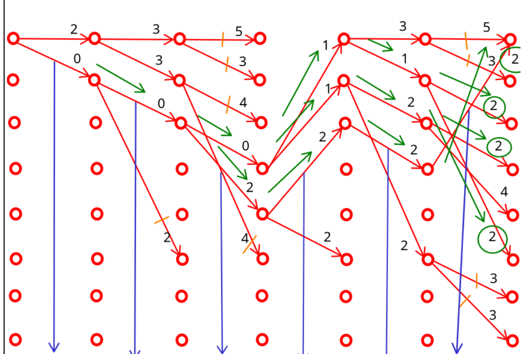
Viterbi algorithm / Decoding convolutional code



now check what the output was with the values of the changes above the 1/10... 0/00...

consider the output from above with 2 errors: 110111101111

the numbers on the line represent the difference to the code given ->Hamming Distance



1/11 0/01 1/00 0/10 0/11 0/11

!!corrected error!!

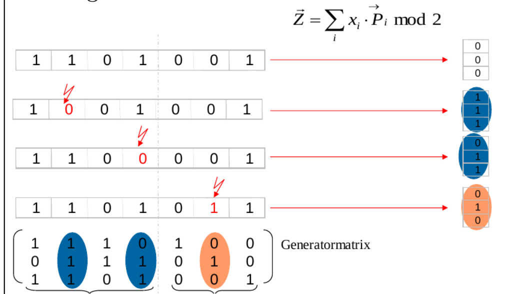
-> 101000

There is a big problem with this algorithm, as you can see there were multiple ways of solving it, and the others would NOT have given the same code. So multi error can still be tricky.

Input - Output ratio for convolutional codes

$R = \frac{input}{output}$ in example $R = \frac{1}{2}$

hamming code extension



message with code check

1101	001
1 x 101 ->	1 x 101
1 x 111	1 x 111
0 x 110	1 x 011
1 x 011	1 x 001
0 x 100	000
0 x 010	correct
1 x 001 ->	

message with code check

1111	001
1 x 101 ->	1 x 101
1 x 111	1 x 111
1x 110	1 x 110
1 x 011	1 x 011
0 x 100	1 x 001
0 x 010	$\boxed{110}$
1 x 001 ->	wrong

error at 110

RSA: !!Check Github for another PDF!!

RINKEL DONE

due to space constraints, something form the last page:

Floating Point Comparison

Due to the limitation of floating point numbers, checking for equality the regular way doesn't make sense

```
for (int = 0; i < n; ++i) { a += f; }
b = ((float) n) * f;
if (a == b) { ... }
```

This checks bit pattern!!

Instead we check if it is approximately equal:

```
if (Math.abs(a - b) <= eps) { ... }
```

Problem with this: it's the absolute error instead we can choose the relative error:

```
float aa = Math.abs(a);
float bb = Math.abs(b);

if (Math.abs(a - b) <= eps * Math.max(a, b))
```

The Max makes sure the eps is based on the bigger number.

There are still certain problems: calculations close to 0 might result in the relative error of 1, which is actually not an error once again rounding errors....

BE AWARE OF FLOATING ERRORS!!

potence system

decimal binary and hexadecimal are all built the same way

$W_a(z) = \Sigma_{i=0}^{n-1} z_i * a^i = z_{n-1} * a^{n-1} + z_{n-2} * a^{n-2} + \dots + z_1 * a^1 + z_0 * a^0$

Terms in the binary form

Bit = 1 set bit

Bit = 0 cleared bit

LSB Least Significant Bit, rightmost bit, bit 0

MSB Most Significant Bit, leftmost bit, bit (n-1)
!!Note that this is for little-endian!!
big-endian is the opposite!!

Nibble a 4 bit binary number
bigger numbers get grouped into nibbles -> 1101 0101

Octett a binary number with 8 bit

byte 8 bit

Note, in binary the leading 0s will still be written, in order to indicate how big the memory allocation is.

indices in bits in order to change single bits in a number, we can use the array notation. b=1010 b[3]=1 b=1011

amount of numbers to display
binary: 2^n numbers -> n=bits
The amount of numbers to double every increase of potence.
Note: There are no broken binary numbers. It's always 2^n with n being an integer!!

prefixes

2^{10}	$1.024 \cdot 10^3$	K	Kilo	Ki	Kibi
2^{20}	$1.049 \cdot 10^6$	M	Mega	Mi	Mebi
2^{30}	$1.074 \cdot 10^9$	G	Giga	Gi	Gibi
2^{40}	$1.100 \cdot 10^{12}$	T	Tera	Ti	Tebi
2^{50}	$1.126 \cdot 10^{15}$	P	Peta	Pi	Pebi
2^{60}	$1.153 \cdot 10^{18}$	E	Exa	Ei	Exbi

suffixes: binary:1011_b, hex:1011_h

Konvention	Dezimal	Hexadez.	Binär
Intel Manual	1011	1011H	1011B
Wikipedia	1011_{10}	1011_{16}	1011_2
C, C++, Java, Python, PHP	1011	0x1011	0b1011
Pascal, Delphi	1011	$1011	%1011
Ada	1011	16#1011#	2#1011#
Visual Basic	1011	&H1011	&B1011

Hex

1 hex decimal is a Nibble, 4 bit
2 hex decimals are a byte
this is why bits are grouped into nibbles and bytes.
Notation: 12 34 -> 0001 0010 0011 0100
just like with binary, we write the leading 0s for the memory size.

biggest number possible with n bit:

		9 Bit	1FF	13 Bit	1FFF
⋮		10 Bit	3FF	14 Bit	3FFF
7 Bit	7F	11 Bit	7FF	⋮	
8 Bit	FF	12 Bit	FFF		

Logical Operations

It is a function from n bits to 1 bit -> bool

$f : \mathbb{B}_n \text{ -> } \mathbb{B} \text{ || } f :: [a] \text{ -> } Bool$

parameter: variable to use inside the function
argument: specific value used in a specific use
can also be described as fixed values. Note f(a,b) = a + g(a,b) in this function the a and b in f are parameters, but in g they are arguments !!

Unary function: a function with 1 parameter
Binary function: a function with 2 parameters
Ternary function: a function with 3 parameters
n-ary function: a function with n parameters
Nullary function: function with zero parameters

There are exactly 4 unary functions (with bits!)
F(X) = 0 -> False || T(X) = 1 -> True
id(x) = x, just like haskell id...
not(x) = $\cancel{x}$

There are 16 binary functions
Disjunction: OR $x \vee v$
Conjunction: AND $x \wedge v$
The others are made with unary, nullary or the 2 above

x	y	0	xy	$x\bar{y}$	x	$\bar{x}y$	y	$x\bar{y} \vee \bar{x}y$	$x \vee y$	$\overline{x \vee y}$	$xy \vee \bar{x}\bar{y}$	$\bar{y}$	$x \vee \bar{y}$	$\bar{x}$	$\bar{x} \vee y$	$\overline{xy}$	1
0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
0	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	0	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1

x	y	0	AND	$x\bar{y}$	x	$\bar{x}y$	y	XOR	OR	NOR	EQUIV	$\bar{y}$	$y \to x$	$\bar{x}$	$x \to y$	NAND	1
0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
0	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	0	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1

note that some are actually unary or nullary, aka they don't actually do anything with the parameters.

NAND: the basis of modern computers
it is easy to create with transistors.
It only results in False if both inputs are true, hence the name Not-AND.
mathematical notation: $NAND = x|y = \overline{x \wedge v}$
all base operations can be made with NAND

$$\overline{x} = \overline{x \wedge x} = x \mid x$$
$$x \wedge y = \overline{x \mid y} = (x \mid y) \mid (x \mid y)$$
$$x \vee y = \overline{\overline{x} \wedge \overline{y}} = (x \mid x) \mid (y \mid y)$$

NOR
NOR is only true when neither of the inputs are true.
aka NOR = 1 if x==0 && y==0
$\overline{x \vee v}$
Just like with NAND, all functions can be made with NOR

XOR: exclusive or
The XOR is true if only one input is true.
$x \oplus y$

Addition of bits.

x	y	$x+y$	$x \wedge y$	$x \oplus y$
0	0	00	0	0
0	1	01	0	1
1	0	01	0	1
1	1	10	1	0

AND signifies the overflow of bits. 1 0 <- 1 1
XOR signifies the addition of bits. 0 OR 1

Literal: variable or negation of variable: $x_1, \overline{x_2}$
Conjunction-term: conjunction of literals $x_1 x_2 = x_1 \wedge x_2$
Disjunction-term: disjunction of literals $x_1 \vee x_2$
Minterm: conjunction with ALL parameters of a function
Maxterm: disjunction with ALL parameters of a function

Disjunctive Normalform DNF
a disjunction of conjunctions. $x_1 x_2 \vee \overline{x_1 x_2}$
Functions are often displayed as DNF, since this format requires only 3 symbols: $\vee \wedge \neg$
The canonical DNF C-DNF: a DNF with all parameters
the canonical DNF is often used to display the true false table.
The C-DNF is then often simplified to get the end result

x	y	$x \oplus y$	$= \overline{x}y \vee x\overline{y}$	$x\|y$	$= \overline{xy} \vee \overline{x}y \vee x\overline{y}$
0	0	0		1	$\overline{xy}$
0	1	1	$\overline{x}y$	1	$\overline{x}y$
1	0	1	$x\overline{y}$	1	$x\overline{y}$
1	1	0		0	

$$\overline{xy} \vee \overline{x}y \vee x\overline{y} = \overline{x}(\overline{y} \vee y) \vee x\overline{y} = \overline{x} \vee x\overline{y} = \overline{x} \vee \overline{y}$$
$$x \oplus y = \overline{x}y \vee x\overline{y}$$

note that DNF has nothing to do with the ferrari engine.

logical function only return 1 or 0. In order to return an entire number, we would have to map this function.
luckily there are several predefined mapped functions
– nor(x) invert all bits in x
– and(x,y) check the individual bits of x,y with and
– or(x,y) check the individual bits of x,y with or
– nor(x,y) check the individual bits of x,y with nor

Bitwise operations in java...

NOT:	$z =$		~	q	$z_i \leftarrow \overline{q}_i$
AND:	$z =$	q	&	p	$z_i \leftarrow q_i \wedge p_i$
OR:	$z =$	q	|	p	$z_i \leftarrow q_i \vee p_i$
XOR:	$z =$	q	^	p	$z_i \leftarrow q_i \oplus p_i$

|| and && are evaluations
note that with || and &&, if the first evaluation is enough to determine the result, the second one won't be executed.
a=False, b=True -> a && b -> a is false, therefore result false.

Variable sizes in java
int = 32 bit
short = 16 bit
byte = 8 bit
long 64 bit
Please note that these should only be used when, you either save significant memory, or the integer isn't big enough.

multiplying a binary number
you have to multiply every single bit and add the corresponding 0s

2^2 101011
↓
4
100 · 101011
000000
0000000
10101100

Multiplication of binary number
$$b = 2^{n-1} \cdot b_{n-1} + \cdots + 2^0 \cdot b_0$$
Multiplication with potences. -> also binary!
$$\begin{aligned} c &= 2^m \cdot (2^{n-1} \cdot b_{n-1} + \cdots + 2^0 \cdot b_0) \\ &= 2^m \cdot 2^{n-1} \cdot b_{n-1} + \cdots + 2^m \cdot 2^0 \cdot b_0 \\ &= 2^{m+n-1} \cdot b_{n-1} + \cdots + 2^m \cdot b_0 \\ &= 2^{m+n-1} \cdot c_{m+n-1} + \cdots + 2^m \cdot c_m + 2^{m-1} \cdot 0 + 2^0 \cdot 0 \end{aligned}$$
$$c_{m+n-1} = b_{n-1}, \ldots, c_m = b_0, c_{m-1} = \cdots = c_0 = 0$$

Multiplication with potences is just a leftshift!!
$2^4 * 101$ = add 4 0s to 101 -> 101'0000

Division with potences is a rightshift!!
$\frac{1'0111}{2^3}$ = remove 3 bits from 1'0111 -> 101

Left & Rightshift java
logical right: a >>> x -> 101 >>> 1 -> 010
logical left: a <<< x -> 101 <<< 2 -> 1'0100
arithmic right: a » x -> 101 » 1 -> 110
arithmic left: a « x -> 101 « 2 -> 1'0111
the »> and «< add the MSB value instead of 0
The reason for this is unsigned and signed!!

Reading a bit
$$\begin{aligned} b \wedge m &= b_7 b_6 b_5 b_4 b_3 b_2 b_1 b_0 \wedge m_7 m_6 m_5 m_4 m_3 m_2 m_1 m_0 \\ &= b_7 b_6 b_5 b_4 b_3 b_2 b_1 b_0 \wedge 0100'0000 \\ &= 0 b_6 00'0000 \end{aligned}$$
Java `b & 0b0100_0000`

Setting a bit
$$\begin{aligned} b \vee m &= b_7 b_6 b_5 b_4 b_3 b_2 b_1 b_0 \vee m_7 m_6 m_5 m_4 m_3 m_2 m_1 m_0 \\ &= b_7 b_6 b_5 b_4 b_3 b_2 b_1 b_0 \vee 0100'0000 \\ &= b_7 1 b_5 b_4 b_3 b_2 b_1 b_0 \end{aligned}$$
Java `b v 0b0100_0000`

Deleting a bit
$$\begin{aligned} b \wedge m' &= b_7 b_6 b_5 b_4 b_3 b_2 b_1 b_0 \wedge m'_7 m'_6 m'_5 m'_4 m'_3 m'_2 m'_1 m'_0 \\ &= b_7 b_6 b_5 b_4 b_3 b_2 b_1 b_0 \wedge 1011'1111 \\ &= b_7 0 b_5 b_4 b_3 b_2 b_1 b_0 \end{aligned}$$
Java `b & 0b1011_1111`

Combine these with right and left shift!
read: b & (1 « n) »> n 000(b AND 11111...)
set: (b | (1 « n)) b OR 11111...
delete: b & (1 « n) b AND not 11111..

given a binary number c and a bit amount p. shift c by p.
1. create mask, 2. read n bits from c into bb
3. shift n bits from c to cc in order to create room for p bits, 4. set bb in cc

```
    m = (1 << p) - 1;   //
    bb = b & ~(m << n); //
    cc = (c | m) << n;  //
    r = bb | cc;        //
```

Addition and Subtraction
Addition:

$$\begin{array}{rccccc} & & 1 & 1 & 1 & 1 \\ + & {}_1 & 0_1 & 0_1 & 0_1 & 1 \\ \hline & (1) & 0 & 0 & 0 & 0 \end{array}$$

Subtraction: (Zweierkomplement)

$$\begin{array}{rcccc} & 1 & 1 & 0 & 1 \\ - & 0 & 0_1 & 1 & 0 \\ \hline & 1 & 0 & 1 & 1 \end{array}$$

if the subtraction would result in 0, then we add another 1 to the number above!!

$$\begin{array}{rccccc} & (1) & 0 & 0 & 0 & 0 \\ - & (0)_1 & 0_1 & 0_1 & 0_1 & 1 \\ \hline & (0) & 1 & 1 & 1 & 1 \end{array}$$

Signed & Unsigned
Unsigned: only positive integers!
Signed: MSB 0 = positive, MSB 1 = negative
the rest is a regular binary number.
note: This is something you simply need to know.
It isn't included in some encoding!!

$$\begin{array}{rccccc} & (1) & 0 & 0 & 0 & 0 \\ - & (0)_1 & 0_1 & 0_1 & 0_1 & 1 \\ \hline & (0) & 1 & 1 & 1 & 1 \end{array}$$

$$N(1) = 2^n - 1 = \underbrace{1 \cdots 1}_{n}$$

$$\begin{array}{rccccc} & (1) & 0 & 0 & 0 & 0 \\ - & (0)_1 & 0_1 & 1_1 & 1_1 & 1 \\ \hline & (0) & 1 & 0 & 0 & 1 \end{array}$$

$$\begin{aligned} N(2^{n-1} - 1) &= 2^n - (2^{n-1} - 1) \\ &= 2 \cdot 2^{n-1} - 2^{n-1} + 1 \\ &= 2^{n-1} + 1 \\ &= 1\underbrace{0 \cdots 0}_{n-2}1 \end{aligned}$$

$N(b) = 2^n - b$ = $N(b) = 0 - b$
this is because of the overflow. see above!!
0 = 0000... -> 10000...

	signed	unsigned
4 Bit	$-8 \ldots 7$	$0 \ldots 15$
8 Bit	$-128 \ldots 127$	$0 \ldots 255$
16 Bit	$-32K \ldots 32K - 1$	$0 \ldots 64K-1$
32 Bit	$-2G \ldots 2G - 1$	$0 \ldots 4G - 1$

note the difference in positive and negative in the signed category! size constraints!

	$0000'0000_b$	$\cdots$	$0111'1111_b$	$1000'0000_b$	$\cdots$	$1111'1111_b$
unsigned	0	$\cdots$	127	128	$\cdots$	255
signed	0	$\cdots$	127	**-128**	$\cdots$	**-1**

Note -1 is 1111'1111. Negative numbers are calculated: 0 - number
This means an overflow on unsigned ints will lead to it being 0 again.
On signed ints, it will drop to negative maximum.
Special cases in signed:
2^{n-1} max -> always negative as MSB = 1
Max : 100000000...
0 -> can't be negative as this bit is used for 2^{n-1}

Note that when increasing memory for signed values, you need to use the » operators to copy the MSB.
increasing memory for -1 -> 4 bit to 8 bit
0000'1111 = 15 !! WRONG !! -> 1111'1111 = -1!!

For signed left shift, check if you have spare memory
left shift without checking might result in loss of MSB!
4 bit max: 1001 <<< 2 = 0100 !! prefix changed !!

multiplication: series of left shifts.
1101 * 110 = 11'0100 + 1'1010 = 100'1110
10 -> add 1 zero, 100 -> add 2 zero, get the sum of both
Size increase: max double -> x^2
110 * 110 -> 1100 + 1'1000 = 11'1000 (3 to 6 bits)

$$1111'1111_b \cdot 1111'1111_b = 1111'1110'0000'0001_b \neq 1$$

⇒ signed Multiplikation ≠ unsigned Multiplikation
check if both operands are negative, if so invert them to positive.
do unsigned multiplication
if only one operand was negative, take the negative result.
⇒ $N_n(a) \cdot N_n(b) = a \cdot b$ und $N_n(a) \cdot b = N_{2n}(a \cdot b) = a \cdot N_n(b)$ (mit $a, b \geq 0$)

Note that we don't need to care about signed, if we do not overwrite the MSB!

Division
should be avoided, slow operation compared to others
32 bit -> 20 times as long as multiplication
64 bit -> 80 times as long as multiplication
– can be replaced with right shift for potences
see /10 for decimal numbers.
signed and unsigned division are completely different

- Unsigned: Iteratives Verfahren für $i = n - 1$ bis $i = 0$:
  - Man überprüft ob $b \cdot 2^i$ in die $n - i$ obersten Bits von a «passt»
  - Wenn ja, dann setzt man im Ergebnis Bit i und zieht $b \cdot 2^i$ von a ab

Inversion
$$N(b+1) > 2^n - 1 - b > 2^n == 0$$
$$0 - 1 = -1 > 11111... > -1 - b = \overline{b}$$
Okay, the idea is that -1 is the number with all bits set to 1
This means that no matter what you do, you can't have overflow.
In fact this means that b can be inverted by subtracting it to -1.

this is -1 !!

-1	1 1 1 1 1
b	- 0 1 1 1 0
$\overline{b}$	1 0 0 0 1

Unsigned in java
You can't declare unsigned integer etc, instead you just use the unsigned functions.
`compareUnsigned`, `divideUnsigned`, `remainderUnsigned`
for `Integer`, `Short`, `Byte`, `Long`

Java comparators
`==, !=, <, >, <=, >=`
- `a != b ↔ !(a == b)`
- `a >= b ↔ !(a < b)`
- `a <= b ↔ (a < b) | (a == b)`
- `a > b ↔ !(a <= b) ↔ !((a < b) | (a == b))`

`we only need == <`

Beause processors are super fast in addition an subtraction, we can simplify equality checks by using these 2 operations.
– example unsigned, check if a < b:
`==, !=, <, >, <=, >=`
- `a != b ↔ !(a == b)`
- `a >= b ↔ !(a < b)`
- `a <= b ↔ (a < b) | (a == b)`
- `a > b ↔ !(a <= b) ↔ !((a < b) | (a == b))`

`we only need == <`

c is the carry bit, and it is only set to 1 if a < b!!
– example signed:

case 1, overflow(wrong prefix)

$(+a) - (-b) = (-d)$	$0111 - 1000 = (1)1111$	$7 - (-8) = -1$
$(-a) - (+b) = (+d)$	$1000 - 0111 = 0001$	$(-8) - 7 = 1$

$\boxed{o = \bar{s}_a s_b s_d \vee s_a \bar{s}_b \bar{s}_d}$ S = MSB of variable.

The o stands for overflow, however it only checks for an overflow of the prefix. Aka it checks wether or not the prefix makes sense.
when we subtract something negative then we expect a positive outcome, which we DIDN'T get above!!

case 2 correct prefix

1. result is positive -> Sd = 0 -> a>b
2. result is negative -> Sd = 1 -> a<b

$\boxed{a < b \rightarrow o \oplus s_d = 1}$

– the check for all 0 or 1 is also fast -> AND...
In java we only work with the signed interpretation by default
use the before mentioned special functions for unsigned!

Character Encoding
A character is encoded via a bijectional function
$$E'(z_0..z_{n-1}) = f(E(z_0)..E(z_{n-1}))$$
The formula must obviously be known by both parties, as the text can't be decoded otherwise.

ASCII
$2^7 = 128 -> 7bit$
– uses both printable characters and nonprintable control characters
– introduction 1963, enacted by US president 1968

33 Control characters
10 decimals 0 - 9 33 special symbols ! " # ...
26 uper case letters A - Z 26 lower case letters a - z

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	SOH 1	STX	ETX	EOT	ENQ	ACK	BEL	BS	TAB	LF	VT	FF	CR	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SB	ESC	FS	GS	RS	US
2	␣	!	"	#	$	%	&	’	(	)	*	+	,	−	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	‘	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

00_h \0 NULL: marker for End of text
08_h \b Backspace
09_h \t Tabspace
$0A_h$ \n Line Feed, Newline
$0D_h$ \r Carriage Return

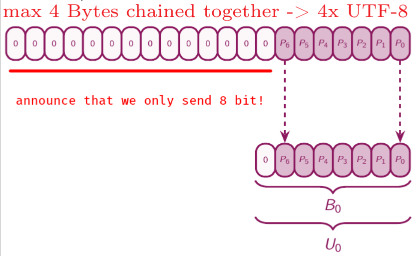
Problems with ASCII
– Just another standard of many, even if it's the first
– Egnlish only, other languages can't to be encoded in ASCII
-> hence no ä ö ü ...

Extensions for ASCII / Codepage
ASCII can be extended since it only uses 7 bit of the available 8. This means that the 8th bit can be used for other characters like ä.
range: 80_h to FF_h
Extension announced with MSB = 1
No extension MSB = 0
Extended ASCII remains compatible to regular ASCII
Many different Codepages defined in ISO.
Codepages have the problem that the sender and receiver need to again know what codepage is being used, since otherwise the receiver might not be able to decode certain characters properly.

Unicode
UTF-8 (Standard) UTF-16 UTF-32
max 17 · 64K - 2K = 1M + 62K = 1,112,064 Code-Points
work in progress

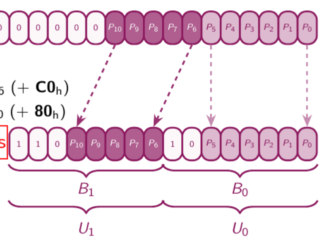
Codepoints(CP): Number of a character
CodeUnit(CU): Unit for displaying encoded character
UTF-32 – all CP displayed with one CU
UTF-16 – all CP displayed with 1-2 CUs
UTF-8 – all CP displayed with 1-4 CUs
UTF-16 and 32 need to check for endianness since they are above 8 bit.
UTF-32 is limited to 21 bit!!
Rest often used for other things.

UTF-8
UTF-8 it itself another ASCII extension
UTF-8 -> 7 bit for characters
max 4 Bytes chained together -> 4x UTF-8 block



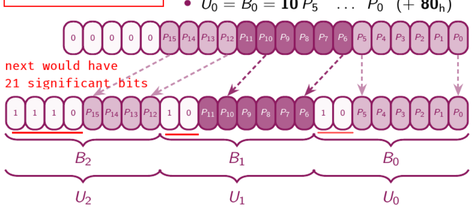
Codepoint in [80, 7FF]:
- $U_1 = B_1 = \mathbf{110}\, P_{10} \dots P_6\ (+\ \mathbf{C0_h})$
- $U_0 = B_0 = \mathbf{10}\, P_5 \dots P_0\ (+\ \mathbf{80_h})$

11 significant bits

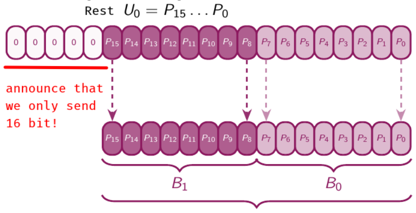


16 significant bits

Codepoint in [800, FFFF]
- $U_2 = B_2 = \mathbf{1110}\, P_{15} \dots P_{12}\ (+\ \mathbf{E0_h})$
- $U_1 = B_1 = \mathbf{10}\, P_{11} \dots P_6\ (+\ \mathbf{80_h})$
- $U_0 = B_0 = \mathbf{10}\, P_5 \dots P_0\ (+\ \mathbf{80_h})$

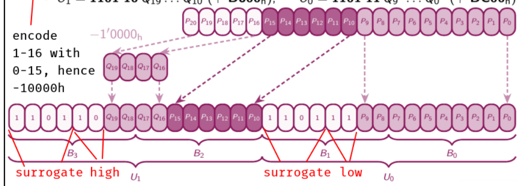


UTF-16
max 4 Bytes chained together -> 2x UTF-16 block
CP in [0, FFFF] without [D800, DFFF]
[D800, DFFF] is not valid
Rest $U_0 = P_{15} \dots P_0$



CP in [1'0000, 10'FFFF] 0 not used here!! therefore - 10000h !!!
- $Q = P - 1'0000_h$ (-> Q in [0, F'FFFF] and $Q_{15} \dots Q_0 = P_{15} \dots P_0$)
- $U_1 = \mathbf{1101'10}\, Q_{19} \dots Q_{10}\ (+\ \mathbf{D800_h})$; $U_0 = \mathbf{1101'11}\, Q_9 \dots Q_0\ (+\ \mathbf{DC00_h})$

encode 1-16 with 0-15, hence -10000h



- The entire block is always interpreted as Big-Endian!!
  - UTF-16BE: Big Endian
    - CP with 1 CU: $U_0 = B_1 B_0$
    - CP with 2 CUs: $U_1 U_0 = B_3 B_2 B_1 B_0$
  - UTF-16LE: Little Endian
    - CP with 1 CU: $U_0 = B_0 B_1$
    - CP with 2 CUs: $U_1 U_0 = B_2 B_3 B_0 B_1$

$U_1 (D800 \leq U_1 \geq DBFF)$ is the high surrogate
$U_0 (DC00 \leq U_0 \geq DFFF)$ is the low surrogate
codepoints above 10000_h are supplementary codepoints
codepoints c with $p0000_h \leq c \leq pFFFF_h$ are on plane p
p = 0 is the basic multilingual plane -> 1 UTF-16 block
$1 \leq p \leq 16$ are the supplementary planes

UTF-32
first 11 bits used for other things
Big-Endian by default -> convert for little endian
21 significant bits
max 4 bytes -> ONLY 1 UTF-32 block
Most used internally in programming languages and or applications.

Byte-Order-Mark (BOM) $FEFF_h$
If this is at the start of a file, then you can read the encoding from it:

UTF-16BE FE FF
UTF-16LE FF FE
UTF-8 EF BB BF
UTF-32BE 00 00 FE FF
UTF-32BE 00 00 FE FF

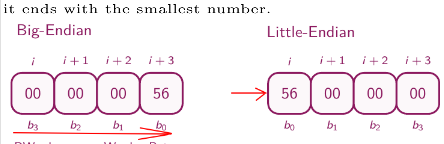
Encoding examples

	CP	UTF-32BE	UTF-32LE	UTF-8	UTF-16BE	UTF-16LE
A	41	00 00 00 41	41 00 00 00	41	00 41	41 00
ä	E4	00 00 00 E4	E4 00 00 00	C3 A4	00 E4	E4 00
α	3B1	00 00 03 B1	B1 03 00 00	CE B1	03 B1	B1 03
ặ	1EB7	00 00 1E B7	B7 1E 00 00	E1 BA B7	1E B7	B7 1E
𐌰	10330	00 01 03 30	30 03 01 00	F0 90 8C B0	D8 00 DF 30	00 D8 30 DF

Enian(ness)
Big-endian and Little-endian
Big-endian -> MSB leftmost bit, LSB rightmost bit
Little-endian -> MSB rightmost bit, LSB leftmost bit

Important since for example Intel processors have 1 Byte as the smallest adressable unit. This means you need to know the Endian of that byte, as you otherwise wouldn't be certain if you hit the LSB or the MSB first.

Endian Problem
Little-endian is objectively better
example: accessing the first 8 bits from a 16 bit number. It would make sense to only request the 8 bits, but this doesn't work for big Endian as it ends with the smallest number.



For little endian, the access could have been smaller -> 8 bit.
Thank fibonacci for fucking up his translation of numbers.

Endian Problem solution
– shift to Mixed or Middle endian
– Acceptance of both Endians
These solutions are needed since it is unlikely that there will unity with endianness.

Encoding in Java
Char receives a UTF-16 Code Unit
Strings are an object with length and CodePoints
This is different from C where you have to iterate to find the NULL symbol 0x00. Java uses more memory but it is faster to check length etc.

```
char c = '1'; // char literal, in UTF-8: 0x31
char d = '\u0031'; // Unicode codepoint literal
char e = 1; // int literal converted to char 0x0001
```

charAt(i)	codePointAt(i)
chars	codePoints
length	codePointCount
charAt(i-1)	codePointBefore(i)
charAt(i+n)	offsetByCodePoints(i, n)

Certain operations are not permitted on Strings directly such as + - delete, insert, replace etc as they would change the string, and a string is immutable!
To mitigate this problem, use buffers and Stringbuilder

ByteBuffer, CharBuffer, IntBuffer
contains: array, capacity, limit and pos
– array is the storage for the type -> char, int
– capacity is the size of the entire buffer, it is immutable
– limit is the amount of changeable elements inside the buffer
– pos is the relative position for read and write operations

- byte get(): get array[pos], increment pos by 1
- get(byte[] d, int s, int c)
  copy array[pos]-array[pos+c-1] into d[s]-d[s+c-1]
  pos += c
  -> exception when pos+c >= limit
- get(byte[] d) = get(d, 0, d.length)
- put(byte b)
  put(byte[] d, int s, int c) same as get, inverted.
  put(byte[] d) = put(d, 0, d.length)

```
while (hasData)
{
  prepareData(...); // calls buffer.put(...);
  buffer.flip(); limit = pos -> pos = 0 ->
//tell the buffer this is the end of the data.
//We can now iterate through the data again from the
//beginning. In other words, end of write, start read.
  processData(...); // calls buffer.get()
  buffer.clear(); clear the buffer for reuse.
}
```

InputStreams
Sequential processing of bytes
int read() -> 0-255 per byte
int read(byte[] d, int s , int c) amount of read bytes
int read(byte[] d)
all of these function return -1 for exceptions

FileInputStream
reads data via functions from the operating system.
should only be used in combination with read(byte[], ...) as it would be extremely slow otherwise.

BufferedInputStream
reads data form another Stream and bufferes it.
very fast read()
if size is unknown then other streams should be used.

```
var stream = new BufferedIS(new FileIS("..."));
```

InputStreamReader
read form an InputStream
has to convert bytes into chars
It uses a decoder d as specified:

```
ISR(IS s, CharsetDecoder d):
ISR(IS s, Charset c): Wie ISR(s, c.newDecoder())
ISR(IS s, String n): Wie ISR(s, Charset.forName(n))
ISR(IS):like ISR(s, Charset.defaultCharset())
```

Charsets in Java
Charset.newDecoder() or Charset.newEncoder()
create decoder or encoder.
static SortedMap<String,Charset> Charset.availableCharsets()
returns all available Charsets for this VM.

FileReader (extends FileInputStream)
– just calls super(FileInputStream) – only available with default en
– for more encoders you have to call FileInputStream manually

BufferedReader
– reads data blockwise from a reader and buffers it
check BufferedInputStream for similarity
String readLine(): gives all chars until next line

```
var reader = new BufferedReader(new FileReader("..."));
```

Streams and Resource-Leaks
Because most Streams need resources from the OS, we need to close these streams again. Otherwise leak!
The try is for possible exceptions, obviously

```
FileReader fr = new FileReader(path);
BufferedReader br = new BufferedReader(fr);
try {
    return br.readLine();
} finally {
    br.close();
    fr.close();
}
```

or the simplified and more safe way (Try-With-Resources): (em.close can cause exception!)

```
try (
    FileReader fr = new FileReader(path);
    BufferedReader br = new BufferedReader(fr)
)
{
    return br.readLine();
}
```

– types have to extend the closeable type.
– resources are automatically closed in the opposite manner
– try with resources guarantees safe closing.

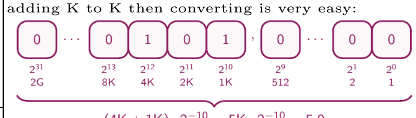
Units in Programming languages Obviously there is no typ
We therefore use comments to signify the km or kg for humans
Keep in mind that this means we also have to implement the conversion ourselves, as the computer doesn't know you want to calculate meters + kilometers!!

```
// in km/h  int length1 = 10; // in cm
int speed;  int length2 = length1 * 10; // in mm
            int tempC = 20; // in C
// in CHF   int tempF = tempC * 9 / 5 + 32; // in F
int amount; Doesn't work!! TRUNCATE!!
```

Fixed Point Numbers: Integers with Commas!

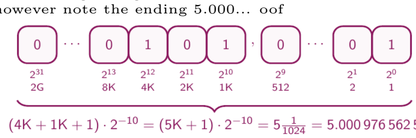
```
int sizeK = 5; // in K
int sizemK = sizeK * (1 << 10); // in mK
```

adding K to K then converting is very easy:



$(4K + 1K) \cdot 2^{-10} = 5K \cdot 2^{-10} = 5.0$

even adding a single bit is not that hard
however note the ending 5.000... oof



$(4K + 1K + 1) \cdot 2^{-10} = (5K + 1) \cdot 2^{-10} = 5\frac{1}{1024} = 5.000\,976\,562\,5$

This can still be displayed, but:

$$5\frac{1}{1024} = 5.000\,976\,562\,5 < 5.001 < 5.001\,953\,125 = 5\frac{2}{1024}$$

5.001 can't be displayed, only the one on the left and right.
Inaccuracy due to integer limitations!
only multiples of 2^{-10} can be displayed

Biggest number in Klotz

$$\underbrace{\boxed{1}_{2^{31}\,2G} \cdots \boxed{1}_{2^{11}\,2K}\ \boxed{1}_{2^{10}\,1K}\ ,\ \boxed{1}_{2^{9}\,512} \cdots \boxed{1}_{2^{1}\,2}\ \boxed{1}_{2^{0}\,1}}$$

$$2^{-10} \cdot \sum_{k=0}^{31} 2^k = 2^{-10} \cdot (2^{32} - 1) = 2^{22} - 2^{-10} = 4M - \frac{1}{1K}$$

Note that $\Sigma_{k=0}^{3} 1$ due to Integer being max 32 bit!

Cut Numbers after comma

$$\underbrace{\boxed{1}_{2^{n-1}} \cdots \boxed{1}_{2^{0}}}\ ,$$

$$2^0 \cdot \sum_{k=0}^{n-1} 2^k = 2^n - 1$$

Cut Numbers before comma

$$,\ \underbrace{\boxed{1}_{2^{n-1}} \cdots \boxed{1}_{2^{0}}}$$

$$2^{-n} \cdot \sum_{k=0}^{n-1} 2^k = 2^{-n} \cdot (2^n - 1) = 2^0 - 2^{-n} = 1 - 2^{-n}$$

Addition: $x \cdot 2^{-a} + y \cdot 2^{-a} = (x + y) \cdot 2^{-a}$

Subtraction: $x \cdot 2^{-a} - y \cdot 2^{-a} = (x - y) \cdot 2^{-a}$

2 complement: (Zweierkomplement) $2^n \cdot 2^{-a} - x \cdot 2^{-a} = (2^n - x) \cdot 2^{-a}$

Shiftoperations: $(x \cdot 2^{-a}) \cdot 2^b = (x \cdot 2^b) \cdot 2^{-a}$

- For multiplication, only the comma position changes

$$(x \cdot 2^{-a}) \cdot (y \cdot 2^{-a}) = (x \cdot y) \cdot (2^{-a} \cdot 2^{-a}) = (x \cdot y) \cdot 2^{-2a}$$

- for a = n ->> $x \cdot y < 2^{2n}$:

$$(x \cdot 2^{-n}) \cdot (y \cdot 2^{-n}) = (x \cdot y) \cdot 2^{-2n} < 2^{2n} \cdot 2^{-2n} = 1$$

Floating Point Numbers

prefix MSB s: 1 bit
Significand z: BitString of number
Exponent a: float -> 8, double -> 11
Encoded Exponent k: explained below
bias b: $-(2^{e-1} - 1)$
precision p: float -> 24, double -> 53
$W((a, z)) = z \cdot 2^a$

Different forms of floating point numbers:

- Single (Java float): 24 bit precision, 8 bit exponent
- Double (Java double): 52 bit precision, 11 bit exponent
- Quadruple (2008): 113 bit precision, 15 bit exponent
- Octuple (2008): not used
- (Single-Extended 43 Bit)
- (Double-Extended 79 Bit)
- (Encodings of base 10 32, 64, 128 Bit, 2008)

Normalized Numbers

$0.75 = 0.110 \cdot 2^0 = \boxed{1.100 \cdot 2^{-1}} = 0.011 \cdot 2^1 = 11.00 \cdot 2^{-2} = (0.)0011 \cdot 2^2 = \ldots$

The normalized number always has the 1 before the comma!!
1.something -> normalized (the one is called the hidden bit)
it is always set, a zero would not be detected!
The hidden bit is NOT an actual bit!!
It is used inside the formula for calculation!

Encoding of Exponents

The bitpatterns 0..0 and 1...1 are not valid exponents
The exponent is encoded as k = a - b, b = $-(2^{e-1} - 1)$
-> $k \in [1, 2^e - 2]$
-> $a \in [-(2^{e-1} - 2), 2^{e-1} - 1] = [-(b - 1), b]$
– for float : $b = 2^{8-1} - 1 = 127 \wedge a \in [-126, 127]$
– for double : $b = 2^{11-1} - 1 = 1023 \wedge a \in [-1022, 1023]$

Value of a floating point number

$$W(x) = (-1)^s * 2^{k+b} * (1 + (z * 2^{-(p-1)}))$$

Example: Note z is the variable here, not the significand
$z = 1 \,|\, 1000\,0000 \,|\, 011\,0000\,0000\,0000\,0000\,0000$

$s = 1 \quad k = 2^7 \quad m = 2^{21} + 2^{20}$

$$\begin{aligned} W(z) &= (-1)^1 \cdot 2^{2^7 + (-(2^7-1))} \cdot (1 + (2^{21} + 2^{20}) \cdot 2^{-23}) \\ &= -2^1 \cdot (1 + (2^{-2} + 2^{-3})) \\ &= -(2 + 2^{-1} + 2^{-2}) \\ &= -2.75 \end{aligned}$$

Special Values:

- $z = 0 \,|\, 0111\,1111 \,|\, 000\,0000\,0000\,0000\,0000\,0000$

$s = 0 \quad k = 2^7 - 1 \quad m = 0$

$$W(z) = (-1)^0 \cdot 2^{2^7 - 1 - (2^7 - 1)} \cdot (1 + 0 \cdot 2^{-23}) = 2^0 = 1$$

- $z = s \,|\, \boxed{kkkk\ kkkk} \,|\, 000\,0000\,0000\,0000\,0000\,0000$

$m = 0$ m = significand

$$\begin{aligned} W(z) &= (-1)^s \cdot 2^{k+b} \cdot (1 + 0 \cdot 2^{-23}) \\ &= \pm 2^{k+b} = \pm 2^a \end{aligned}$$

- $z = 0 \,|\, 1111\,1110 \,|\, 111\,1111\,1111\,1111\,1111\,1111$ = `Float.MAX_VALUE`

$k = 2^8 - 2 \quad m = 2^{23} - 1$

$$\begin{aligned} W(z) &= 2^{2^8 - 2 - (2^7 - 1)} \cdot (1 + (2^{23} - 1) \cdot 2^{-23}) \\ &= 2^{2^7 - 1} \cdot (1 + 1 - 2^{-23}) = 2^{127} \cdot 2 \cdot (1 - 2^{-24}) \\ &= 2^{128} - 2^{104} \end{aligned}$$

- $z = 0 \,|\, 0000\,0001 \,|\, 000\,0000\,0000\,0000\,0000\,0000$ = `Float.MIN_NORMAL`

$$W(z) = 2^{1 - (2^7 - 1)} = 2^{-126}$$

Rounding Modes

mathematical operations are done before rounding!

Round to nearest:

submode 1: Ties to Even: round to an even number, LSB = 0
0.1 -> 0, 1.1 -> 2.0, 3.2 -> 2.0
– cancels itself out over time
– better when numbers with 0.1 are added
– default variant
submode 2: Ties away from Zero
Round to next number 1.1 -> 2.0

Round toward Zero: truncate -> 1.5325 -> 1.0

Round toward +Infinity: Ceiling 1.1 -> 2.0

Round toward -Infinity: Floor, 1.9 -> 1.0

Rounding errors

- $z = 0 \,|\, 0111\,1111 \,|\, 000\,0000\,0000\,0000\,0000\,0001$

$$W(z) = (-1)^0 \cdot 2^{2^7 - 1 - (2^7 - 1)} \cdot (1 + 1 \cdot 2^{-23}) = 1 + 2^{-23}$$

- Die Zahl $z' = 0 \,|\, 0111\,1111 \,|\, 000\,0000\,0000\,0000\,0000\,000\mathbf{0}\,\mathbf{1}$
this has too many bits, we can't display this!!
the precision for float is 23 not 24!

$$W(z') = (-1)^0 \cdot 2^{2^7 - 1 - (2^7 - 1)} \cdot (1 + 1 \cdot 2^{-24}) = 1 + 2^{-24}$$

$\Rightarrow$ The solution in programming languages is to simply round the number. However this is unprecise...
for our number here it would round to z. aka $1 + 2^{-23}$

This error can now be quantified:
first we display a real number x as a floating point number

$$W(x) \to x \in \mathbb{R}$$

The absolut error E is:

$$E(x) = |W(x) - x|$$

The relative error e is:

$$e(x) = \left|\frac{E(x)}{x}\right| = \left|\frac{R(x) - x}{x}\right|$$

For the number from before, an example:

E(z') = $W(z) - W(z') = 1 + 2^{-23} - (1 + 2^{-24}) = 2^{-23} - 2^{-24} = 2^{-24}$

$$e = \frac{W(z) - W(z')}{W(z')} = \frac{2^{-24}}{1 + 2^{-24}} = \frac{1}{2^{24} + 1} < \frac{1}{2^{24}}$$

$z' = 0 \,|\, 0111\,1111 \,|\, 000\,0000\,0000\,0000\,0000\,000\mathbf{0}\ 01$

In this case the number would be rounded down, to 1

Machine Epsilon: ϵ The biggest possible error

$$\epsilon = 2^{-24} = 2^{-2}$$

The smallest number ϵ, so that $1 + \epsilon > 1$
Note: the worst error is therefore always the flip of a highest bit
In this case the flip of 2^{-23} to 1 after 2^{-24} was rounded.

In Programming languages the epsilon is defined as:
the distance from 1 to the next smallest number 2^{-23}
Therefore the biggest error is often
a slight bit bigger than we have defined

$\Rightarrow \varepsilon_c = 2^{-p+1} = 2\varepsilon$

Float and Double in Java

double default when using . -> 0.5
optionally defined with d -> 0.5d
float -> 0.5f f is necessary!

Addition of floating point numbers

1. check prefix, if different do subtraction
2. set prefix for result -> 0 or 1
3. if exponents are different, shift smaller number to right
First shift a single 1 -> hidden bit !
Shift the rest with 0s
4. do addition just like with integers
5. (if carry == 1) normalize result
increment exponent by one
shift significand to the right by 1

$x = 1.5,\ y = 0.75$ (bit) = hidden bit
$x = 0 \,|\, 0111\,1111 \,|\, 100\,0000\,0000\,0000\,0000\,0000$
$y = 0 \,|\, 0111\,1110 \,|\, 100\,0000\,0000\,0000\,0000\,0000$
$x' = 0 \,|\, 0111\,1111 \,|(1)\,100\,0000\,0000\,0000\,0000\,0000$
$y' = 0 \,|\, 0111\,1111 \,|(0)\,110\,0000\,0000\,0000\,0000\,0000 \quad a + 1,\ z \cdot 2^{-1}$
$z' = 0 \,|\, 0111\,1111 \,|(10)\,010\,0000\,0000\,0000\,0000\,0000 = x' + y'$
$z'' = 0 \,|\, 1000\,0000 \,|(1)\,001\,0000\,0000\,0000\,0000\,0000\ a + 1,\ z \cdot 2^{-1}$
$z = 0 \,|\, 1000\,0000 \,|\, 001\,0000\,0000\,0000\,0000\,0000$
$z = 2.25$

Guard digits

– just as hidden as the hidden bit
– implemented in hardware not software
– essentially a flag for rounding!

- $x = 1.999'999'880'790'710'449'218'750 = 2 - 2^{-23}$
- $y = 0.000'000'059'604'644'775'390'625 = 2^{-24}$
- $x = 0 \,|\, 0111\,1111 \,|\, 111\,1111\,1111\,1111\,1111\,1111$
- $y = 0 \,|\, 0110\,0111 \,|\, 000\,0000\,0000\,0000\,0000\,0000$
- $x' = 0 \,|\, 0111\,1111 \,|(1)\,111\,1111\,1111\,1111\,1111\,1111$
- $y' = 0 \,|\, 0111\,1111 \,|(0)\,000\,0000\,0000\,0000\,0000\,0000 \quad a + 24,\ m \cdot 2^{-24}$
- $z' = 0 \,|\, 0111\,1111 \,|(1)\,111\,1111\,1111\,1111\,1111\,1111 = x' + y'$
- $z = 0 \,|\, 0111\,1111 \,|\, 111\,1111\,1111\,1111\,1111\,1111$
- $z = x$
- result is wrong, should be rounded up!!

$x = 1.999'999'880'790'710'449'218'750 = 2 - 2^{-23}$
$y = 0.000'000'059'604'644'775'390'625 = 2^{-24}$
$x = 0 \,|\, 0111\,1111 \,|\, 111\,1111\,1111\,1111\,1111\,1111$
$y = 0 \,|\, 0110\,0111 \,|\, 000\,0000\,0000\,0000\,0000\,0000$
$x' = 0 \,|\, 0111\,1111 \,|(1)\,111\,1111\,1111\,1111\,1111\,1111(0)$ Guard-Digit
$y' = 0 \,|\, 0111\,1111 \,|(0)\,000\,0000\,0000\,0000\,0000\,0000(1)$ Guard-Digit
$z' = 0 \,|\, 0111\,1111 \,|(1)\,111\,1111\,1111\,1111\,1111\,1111(1) = x' + y'$
$z'' = 0 \,|\, 0111\,1111 \,|(10)\,000\,0000\,0000\,0000\,0000\,0000(0)$ rounded
$z = 0 \,|\, 1000\,0000 \,|\, 000\,0000\,0000\,0000\,0000\,0000$

compared to before, not a single digit is the same....
the guard digit did the trick by flagging z to be rounded after addition.

Special cases for floating point numbers ±0 -> a = 0 ..

subnormal number -> a = 0 .. 0 & z NOT 0 .. 0
±∞ -> a = 1 .. 1 & z = 0 .. 0
NaN -> a = 1 .. 1 & z NOT 0 .. 0
The reason we want things such as infinity is this:

```
float x2 = x * x;
if (x2 < max_float) {
    float y2 = y * y;
    if (y2 < max_float) {
        float r = x2 + y2;
        if (r < max_float) {
            r = sqrt(r);
        }
    }
}
```

We either check for overflows the entire time!
Or we just continue calculation with infinity
And then at THE END we check if it's infinite

```
float r = sqrt (x*x + y*y);
if (r == Float.POSITIVE_INFINITY) { ... }
```

Calculation with ∞ exchange ∞ with x and define: lim

if the limit exists, the result is the limit
if the limit doesn't exist, result is NaN
$y \pm \infty = \pm\infty$ >>> $\lim_{x \to \infty} y + x = 0$
$\pm x / 0 = \pm\infty$ for $x \neq 0$
$y / \pm\infty = 0$ for $y \neq 0$ >>> $\lim_{x \to \infty} y / x = 0$
$\sqrt{\infty} = \infty$ >>> $\lim_{x \to \infty} \sqrt{x} = \infty$

Calculation with NaN

ALL OPERATION WITH NaN RESULT IN NaN!
Other operations might also result in NaN:
Z.B. $-\infty + \infty$
Z.B. $\pm\infty \cdot 0$
Z.B. $0/0$ or $\pm\infty / \pm\infty$
Z.B. $\sqrt{x}$ for $x < 0$
There are also some implementations for exceptions inside floating
qNaN: propagates exception
sNaN: throws exception

+0/-0

+0 is the underflow for positive numbers
-0 is the underflow for negative numbers
This is important since certain functions like log
are only defined for positive numbers,
therefore we need a positive 0.
Should we have a rounded down negative number
then we would still want a negative 0
otherwise it would suddenly be considered to be positive.

Subnormal Numbers

No hidden bit! Used when the exponent is too small

$$V(x) = (-1)^s * 2^{-(b-1)} * z * 2^{p-1}$$

Often used as a gradual underflow
or for semi results during a calculation

- $y = s \,|\, 0000\,0000 \,|\, 111\,1111\,1111\,1111\,1111\,1111$

$$\begin{aligned} V(y) &= \pm 2^{-126} \cdot (2^{23} - 1) \cdot 2^{-23} \\ &= \pm 2^{-126} \cdot (1 - 2^{-23}) \\ &= \pm 2^{-126} - 2^{-149} \end{aligned}$$

- $y = 0 \,|\, 0000\,0000 \,|\, 000\,0000\,0000\,0000\,0000\,0001$ = `Float.MIN_VALUE`

$$V(y) = 2^{-126} \cdot 2^{-23} = 2^{-149}$$

Integers as Floating Point Numbers

$$G(x) = 2^{k-b} * 1$$

Regular numbers obviously don't need comma shifting!!
16million + 1 is the smallest number that can't be display
– a=23 only integers can be displayed
– a=24 only even numbers can be displayed
– a=25 only numbers divisible by 4 can be displayed
– a=26 only numbers divisible by 8 can be displayed
– $a \pm 23$ only numbers divisible by 2^{a-23} can be displayed

check page 2 for floating point comparison!