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
- $a^2 = a^2$ aber
- $a^3 = \boldsymbol{a+1}$
- $a^4 = a(\boldsymbol{a+1}) = a^2 + a$
- $a^5 = a(a^2 + a) = a^3 + a^2 = a^2 + \boldsymbol{a+1}$
- $a^6 = a(a^2 + \boldsymbol{a+1}) = a^3 + \boldsymbol{a^2 + a} = \boldsymbol{a+1} + \boldsymbol{a^2} + \boldsymbol{a} = \boldsymbol{a^2} + 1$
- $a^7 = \boldsymbol{a(a^2+1)} = a^3 + a = \boldsymbol{a+1} + a = 1$
- a^8 = a : der Zyklus beginnt von vorne!

- **{0, 1, a, a², a+1, a² + a, a² + a+1, a² + 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

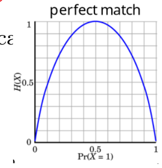
Nachricht (Darstellung & Bedeutung)	redundant	nicht-redundant
irrelevant	Zeichenvorrat bei Quelle und Senke verschieden	
relevant	vorhersagbar	Information

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_s : A3g2beh3f4g => | w_s |= 11

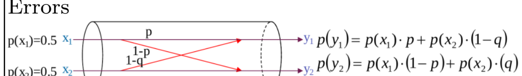
A + 3 x g + 2 x b + e + h + 3 x f + 4 x g

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.

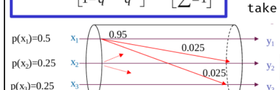
Errors



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

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

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



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

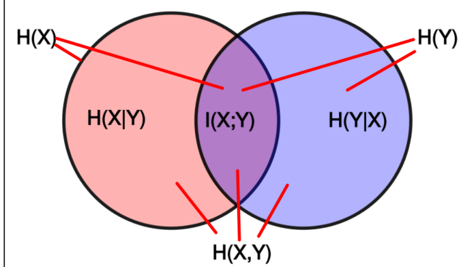
$$p(Y|X) = \begin{bmatrix} 0.95 & 0.025 & 0.025 \\ 0.025 & 0.95 & 0.025 \\ 0.025 & 0.025 & 0.95 \end{bmatrix}$$

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

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
$h = Min_{i,j}(d(x_i, x_j))$

error detection distance
the amount of bits that differ from input to output
$e^* = h - 1$

error correction distance for h even
$h = 2e + 2 -> e = \frac{h-2}{2}$

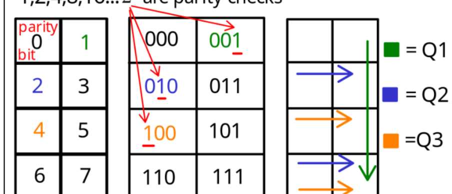
error correction distance for h uneven
$h = 2e + 1 -> e = \frac{h-1}{2}$

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.

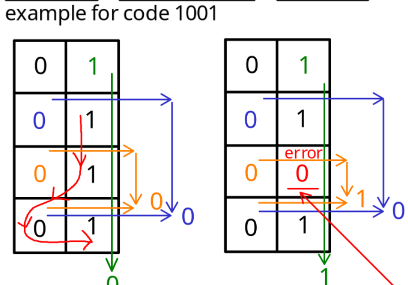
tighly packed coderoom
n = dimension of code
m = dimension of messages $2^m * \Sigma_{w=0}^e \binom{n}{w} \leq 2^n$
k = dimension of control -> n = m + k
The code is considered to be tightly packed
if the equation has the result 2. aka == not smaller.

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=noerror $\vec{Z}$=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:

$par = log_2(\text{bit amount of code})$

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

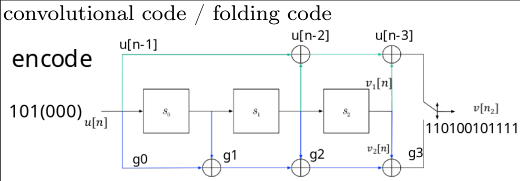
in binary:

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	-	-	-

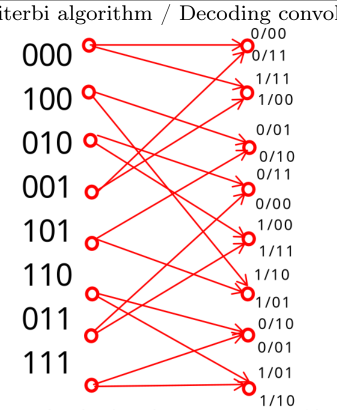
this algorithm is heavily dependent on the configuration, aka the more s states you have the more bits you get as the output.

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

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

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

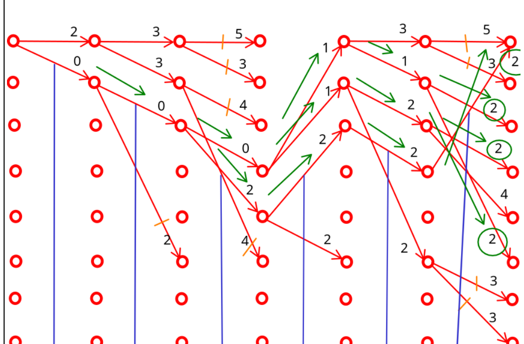
Viterbi algorithm / Decoding convolutional code



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: 1101$\boxed{11}$101111

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



1/11 0/01 $\boxed{1/00}$ 0/10 0/11 0/11

!!corrected error!!

-> 101000

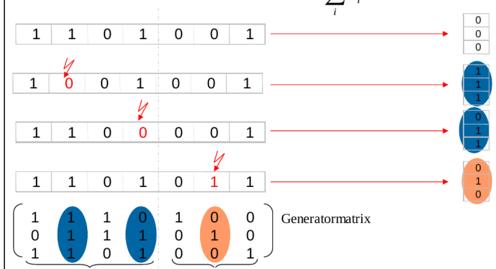
There is a big problem with this algorithm, as you can see there were multiple ways of solving it, and the others would NOT have given the same code. So multi error can still be tricky.

Input - Output ratio for convolutional codes

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

hamming code extension

$\vec{Z} = \sum_i x_i \cdot \vec{P}_i \bmod 2$



RSA

RINKEL DONE

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

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

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: $\wedge \vee \neg$
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	$x \oplus y$	$= \overline{x}y \vee x\overline{y}$	$x\|y$	$= \overline{x}\overline{y} \vee \overline{x}y \vee x\overline{y}$
0	0	0		1	$\overline{x}\overline{y}$
0	1	1	$\overline{x}y$	1	$\overline{x}y$
1	0	1	$x\overline{y}$	1	$x\overline{y}$
1	1	0		0	

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

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
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!
$$c = 2^m \cdot (2^{n-1} \cdot b_{n-1} + \cdots + 2^0 \cdot b_0)$$
$$= 2^m \cdot 2^{n-1} \cdot b_{n-1} + \cdots + 2^m \cdot 2^0 \cdot b_0$$
$$= 2^{m+n-1} \cdot b_{n-1} + \cdots + 2^m \cdot b_0$$
$$= 2^{m+n-1} \cdot c_{m+n-1} + \cdots + 2^m \cdot c_m + 2^{m-1} \cdot 0 + 2^0 \cdot 0$$
$$c_{m+n-1} = b_{n-1}, \ldots, c_m = b_0, c_{m-1} = \cdots = c_0 = 0$$

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
$$b \wedge m = b_7b_6b_5b_4b_3b_2b_1b_0 \wedge m_7m_6m_5m_4m_3m_2m_1m_0$$
$$= b_7b_6b_5b_4b_3b_2b_1b_0 \wedge 0100'0000$$
$$= 0b_600'0000$$
Java `b & 0b0100_0000`

Setting a bit
$$b \vee m = b_7b_6b_5b_4b_3b_2b_1b_0 \vee m_7m_6m_5m_4m_3m_2m_1m_0$$
$$= b_7b_6b_5b_4b_3b_2b_1b_0 \vee 0100'0000$$
$$= b_71b_5b_4b_3b_2b_1b_0$$
Java `b v 0b0100_0000`

Deleting a bit
$$b \wedge m' = b_7b_6b_5b_4b_3b_2b_1b_0 \wedge m'_7m'_6m'_5m'_4m'_3m'_2m'_1m'_0$$
$$= b_7b_6b_5b_4b_3b_2b_1b_0 \wedge 1011'1111$$
$$= b_70b_5b_4b_3b_2b_1b_0$$
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:

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

Subtraction: (Zweierkomplement)

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

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

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

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!!

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

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

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

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

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

	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)

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

$\Rightarrow$ signed Multiplikation $\neq$ 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.

$\Rightarrow$ $N_n(a) \cdot N_n(b) = a \cdot b$ 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	11111
b	-01110
$\overline{b}$	10001

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` $\leftrightarrow$ `!(a == b)`
- `a >= b` $\leftrightarrow$ `!(a < b)`
- `a <= b` $\leftrightarrow$ `(a < b) | (a == b)`
- `a > b` $\leftrightarrow$ `!(a <= b)` $\leftrightarrow$ `!((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` $\leftrightarrow$ `!(a == b)`
- `a >= b` $\leftrightarrow$ `!(a < b)`
- `a <= b` $\leftrightarrow$ `(a < b) | (a == b)`
- `a > b` $\leftrightarrow$ `!(a <= b)` $\leftrightarrow$ `!((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!