Boolean functions

There are 16 possible functions with 2 bits of input and 1 bit of output.
Of these, only 6 are gates:

AND, OR, XOR, NAND, NOR, XNOR

All possible Boolean functions can be written using at most 3 gates:

Set {AND, OR, NOT} is computationally complete.

Also {NAND}, {NOR}, and some others.

Example: use NAND to implement OR

```
x | y = ~~(x | y)
      = ~(~x & ~y)
      = ~x NAND ~y
```

Looks like we also need NOT
However, consider the following:

```
  ~x  = ~x | ~x             a == a | a
      = ~(x & x)            DeMorgan's law
      = x NAND x            Definition of NAND
```

So, `x | y = (x NAND x) NAND (y NAND y)`

A bit ugly, perhaps, but true.

Boolean functions: minterms

Consider a particular truth table with 3 inputs:

row	x_0	x_1	x_2	z	
0	0	0	0	0	
1	0	0	1	0	
2	0	1	0	0	
3	0	1	1	0	
4	1	0	0	0	
5	1	0	1	1	x0\x1x2 == x0 AND ~x1 AND x2
6	1	1	0	0	
7	1	1	1	0	

Want to write a Boolean function for this truth table

Definition: literal is either a Boolean variable (x) or its negation (\x); text uses overbar

We need to write some expressions involving literals for the 3 inputs

Minterm: a term containing exactly 1 instance of each variable, either itself or its complement.

Example: in row 5, `x0\x1x2` has the value 1.

Boolean functions: sum of products

What if more than one output in the truth table is 1?
If m outputs are 1, we need m minterms.

For each row with output 1, construct the minterm.
Combine the minterms by OR operators.

This is called the sum of products.

Products: each minterm is the result of combining literals with AND
Sum: represents combining minterms with OR

Example:

row	x_0	x_1	x_2	z	Minterms
0	0	0	0	0	
1	0	0	1	0	
2	0	1	0	1	\x0x1\x2
3	0	1	1	0	
4	1	0	0	0	
5	1	0	1	1	x0\x1x2
6	1	1	0	0	
7	1	1	1	1	x0x1x2

Function: `z = \x0x1\x2 + x0\x1x2 + x0x1x2`

Boolean functions: sum of products

Example: majority function

Inputs: 3 **Output: 1 whenever more than half of the inputs are true.**

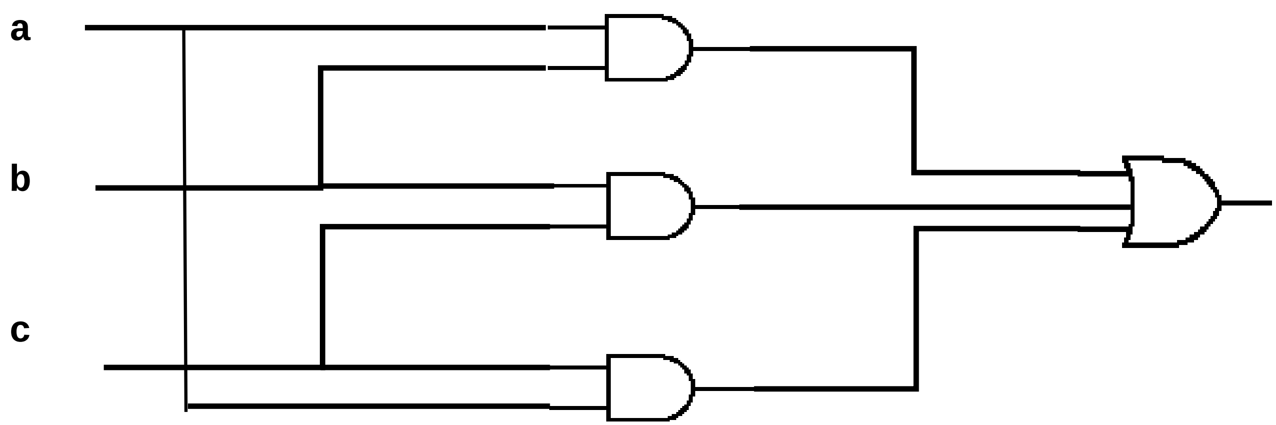
a	b	c	z	Minterms
0	0	0	0	
0	0	1	0	
0	1	0	0	
0	1	1	1	\abc
1	0	0	0	
1	0	1	1	a\bc
1	1	0	1	ab\c
1	1	1	1	abc

```
z = \abc + a\bc + ab\c + abc
```

This can be simplified:

```
z = \abc + a\bc + ab (\c + c)
  = \abc + a\bc + ab
  = \abc + a\bc + ab + abc      Why?
  = bc(a + \a) + a\bc + ab
  = bc + a\bc + ab + abc
  = bc + ac(\b + b) + ab
  = bc + ac + ab
```

Boolean functions: sum of products



Majority function **bc + ac + ab**

Boolean functions: sum of products

Canonical form: can represent any truth table using AND, OR, NOT

- **Sum of products (minterms)**

If output is always 0: `z = 0`

Product of sums

- **Look at rows containing 0**
- **Create maxterms involving sums (OR) of input literals**
- **Use AND to combine the maxterms**

Minimization

- **Techniques are available to:**
  - **reduce the number of minterms**
  - **reduce the total number of literals**
- **Karnaugh maps: graphical method**

Boolean functions: functional completeness

Sum of products can represent any truth table:

$z = p_1 + p_2 \ . \ . \ . + p_n$

each $p_i = l_1 l_2 \ . \ . \ . \ l_m$

and each l_k **is a literal**

Applying double negation to the right hand side,

$z = \sim\sim(p_1 + p_2 \ . \ . \ . + p_n)$

$= \sim(\sim p_1 * \sim p_2 \ . \ . \ . * \sim p_n)$ **by DeMorgan's law**

OR has been eliminated.

Therefore, {NOT, AND} is a functionally complete set.

Similarly, {NOT, OR} is also functionally complete.

Are {AND} and {OR} functionally complete? No.

Consider any Boolean function composed of only these functions.

If all of the inputs are 1, then the output MUST be 1,

and if all the inputs are 0, then the output MUST be 0.

`1 AND 1 == 1, 1 OR 1 == 1       0 AND 0 == 0, 0 OR 0 == 0`

However, it is certainly possible to contruct a truth table where the output

is 0 when all the inputs are 1, and vice versa.

Boolean functions: functional completeness

What about using only 1 Boolean function?

We showed earlier that OR could be implemented using only NAND:

```
x | y = (x NAND x) NAND (y NAND y)
```

In the process of doing so, we also showed that NOT could be implemented with NAND:

```
~x = x NAND x
```

Since {OR, NOT} is functionally complete, so is {NAND}

Similarly, we can show that OR and NOT can be implemented with NOR:

```
~x = ~(x | x)
   = x NOR x
x | y = ~~(x | y)
      = ~(x NOR y)
      = (x NOR y) NOR (X NOR y)
```