Logic

This lecture will deal with logic. There are two reasons for this. First, logic per se is important for a computer scientist; it's worth to handle logic expressions correctly, as they appear e.g. in conditional statements while programming. The second reason is more directly connected to this course: logic expressions are the foundation on which all arithmetic computations in computer processors are based (as we'll describe in more detail in later lectures).

Predicate calculus

Logic deals with the **logical value** (*true* or *false*) of **predicates**. Predicates are essentially *sentences*, though not ones like "Polish mountains are most beautiful", but rather "5 is larger than 6 *or* 3 is an odd number". While other areas of maths let us claim that "3 is odd" is true and "5 is larger than 6" is false, it is logic which will deal with the logical value of these two sentences combined with the "*or*" word.

Logical operators

Let's start from discussing the four basic logical operators:

- AND (**conjuntion**)
- OR (**disjunction**)
- XOR (**exclusive disjunction**; aka. **exclusive or**; meaning: "*either this or that (but not both)*")
- NOT (**negation**)

A predicate a AND b is true *if and only if both* a and b are true. (Note that it's irrelevant here what a and b actually say).

A predicate a OR b is true if and only if *at least one* of a, b is true. That is, they may be both true, or just one of them may be true. In other words, a OR b is false if and only if both these predicates are false. For example, the sentence "tomorrow it will rain *or* tomorrow it will not rain" is always true.

A predicate a XOR b is true if and only if *exactly one* of a, b is true. That is, they *cannot* be both true — exactly one of them has to be. For example, the sentence "I am now in Warsaw XOR I am now in the capital of Poland" is always false (regardless of whether the speaker is actually now in Warsaw or not).

A predicate NOT a means the negation of a, so it's true if and only if a is *false*. For example, the sentence "Warsaw is *not* the capital of Spain" is true. (Should we want the casual syntax to closer reflect the formal syntax of this predicate, we could rephrase it as: "*It's not true that* Warsaw is the capital of Spain").

True and false symbols

As we see, describing logic with words can be troublesome, and take some space. Hence a more concise notation can help. In logic, it's customary to denote "**true**" by **1**, and "**false**" by **0**. Then, what we just said about the AND connective, can be expressed much more directly:

$$0 \text{ AND } 0 = 0, \qquad 1 \text{ AND } 0 = 0, \qquad 0 \text{ AND } 1 = 0, \qquad 1 \text{ AND } 1 = 1.$$

Truth tables

The above form of presentation can be still enhanced. This is achieved by the so-called **truth tables**. Below, rows correspond to the possible logical values of predicate a, and columns to the values of predicate b. Then, each white cell contains the value of a AND b.

AND	0	1
0	0	0
1	0	1

Below, we present analogous tables for the other operators discussed before:

OR	0	1
0	0	1
1	1	1

XOR	0	1
0	0	1
1	1	0

NOT	
0	1
1	0

The truth table for the NOT operator has a bit different shape, as that operator takes just one argument.

With truth tables, one can easily define more logical operators. For example, the following tables define two more commonly used connectives, NAND (**not-and**) and NOR (**not-or**):

NAND	0	1
0	1	1
1	1	0

NOR	0	1
0	1	0
1	0	0

There may be many operators used in one predicate, for example: (a XOR b) AND (a AND NOT b). For such a complex expression, we can also build a truth table. In this case, it would look as follows:

(a XOR b) AND (a AND NOT b)		b	
		0	1
a	0	0	0
	1	1	0

This time (unlike previously), swapping roles of a and b would impact the value of the final expression — and that's why, for full clarity, we indicated in the truth table that rows correspond to the values of a and columns to the values of b.

Of course, there may be more than 2 variables in a predicate, for example:

$$(a \text{ XOR } b) \text{ AND } (a \text{ AND NOT } c).$$

Then, unfortunately, the truth table must look less legibly. The important thing is that it should contain **all possible combinations** of values of the input variables:

a	b	c	$(a \text{ XOR } b) \text{ AND } (a \text{ AND NOT } c)$
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	0

Other notations

In logic, there are various notational conventions in common use. Instead of NOT a, we can write:

$$\text{NOT } a = \neg a = \overline{a} = -a = \sim a$$

Also for other operators, there are a few synonymous ways of writing them down:

$$a \text{ AND } b = a \wedge b = a \cdot b, \qquad a \text{ OR } b = a \vee b = a + b, \qquad a \text{ XOR } b = a \veebar b.$$

Properties of logical operators

So far, we've covered the definitions of logical operators, and the related notation. Then, an interesting question is what relationships hold between them. The basic ones are listed below:

Property name	For AND	For OR
neutral elements	$p \cdot 1 = p$	$p + 0 = p$
extreme elements	$p \cdot 0 = 0$	$p + 1 = 1$
idempotence	$p \cdot p = p$	$p + p = p$
complement	$p \cdot \overline{p} = 0$	$p + \overline{p} = 1$
commutativity	$p \cdot q = q \cdot p$	$p + q = q + p$
associativity	$(p \cdot q) \cdot r = p \cdot (q \cdot r)$	$(p + q) + r = p + (q + r)$
distributivity	$p \cdot (q + r) = (p \cdot q) + (p \cdot r)$, $p + (q \cdot r) = (p + q) \cdot (p + r)$	
absorption	$(p \cdot q) + p = p$, $(p + q) \cdot p = p$	
de Morgan's laws	$\overline{p \cdot q} = \overline{p} + \overline{q}$, $\overline{p + q} = \overline{p} \cdot \overline{q}$	

One can validate these equalities by analyzing the truth tables for both sides and confirming that both sides have equal values, regardless of the input values of p, q and r.

Let us note that the above table contains many well-known properties of standard multiplication and addition (e.g. commutativity, associativity, the first of the distributivity properties etc.). That's why the **convention** of denoting AND, OR, truth and false respectively by $\cdot$, $+$, 1 and 0 is so intuitive. Still, we have to exercise caution, as **not all rules of arithmetics** hold for these symbols interpreted in the world of logic: for example, in logic, we have $1+1 = 1$, which *violates* the property of standard addition that $a + b = a$ necessarily implies that $b = 0$. (On the other hand, comparing to the standard arithmetic, the operations $+$ and $\cdot$ gain some new elegant properties in the world of logic, e.g. the second of the distributivity properties, absorption etc.).

Boolean functions

A **Boolean function** is a function from the set of possible *valuations of Boolean variables* to the set $\{0, 1\}$, where a **Boolean variable** is a variable which can take value 0 or 1, and a **valuation** of a set of variables means a particular assigment of values (0 or 1) to each of them. So, putting it equivalently, a Boolean function is just a truth table.

Notably, as typically in maths, a **function** is just the *assignment* of return values to every choice of argument values, **not the formula** (or any other way) used to describe that assignment. For example, the two formulas below describe **the same function**:

$$f(p, q) = \text{NOT}\,((\text{NOT}\ p)\ \text{AND}\ q), \qquad f(p, q) = (\text{NOT NOT NOT}\ q)\ \text{OR}\ p,$$

because, for *every* choice of values $p, q \in \{0, 1\}$, both formulas produce the same result.

Describing Boolean functions with formulas

An interesting question is how to represent a given Boolean function with a particular formula. To do this, we can start with finding all the input variable valuations for which the function result is 1.

For example, for the following truth table:

a	b	c	$f(a,b,c)$
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

there are exactly 3 valuations leading to result 1: $(0,0,1)$, $(1,0,1)$, and $(1,1,0)$. Now, for each of these valuations, we can easily produce a formula for the function which returns 1 *only* for that valuation:

$$\begin{aligned}
&\text{for } (0,0,1): && (\mathsf{NOT}\ a)\ \mathsf{AND}\ (\mathsf{NOT}\ b)\ \mathsf{AND}\ c &&= \overline{a}\cdot\overline{b}\cdot c\\
&\text{for } (1,0,1): && a\ \mathsf{AND}\ (\mathsf{NOT}\ b)\ \mathsf{AND}\ c &&= a\cdot\overline{b}\cdot c\\
&\text{for } (1,1,0): && a\ \mathsf{AND}\ b\ \mathsf{AND}\ (\mathsf{NOT}\ c) &&= a\cdot b\cdot\overline{c}
\end{aligned}$$

Then, using the operator OR (denoted here by $+$), we can simply build a formula representing our original truth table:

$$f(a,b,c) = (\overline{a}\cdot\overline{b}\cdot c) + (a\cdot\overline{b}\cdot c) + (a\cdot b\cdot\overline{c}).$$

In this way, we can represent every Boolean function. Unfortunately, this does not have to be the shortest possible representation. To see this, consider a function taking 4 input variables which always returns 1 (or, to put it simpler, is always true). The shortest formula describing such a function would be $f(a,b,c,d) = 1$; yet, our general method would produce a disjunction of 16 conjunctions:

$$f(a,b,c,d) = (a\cdot b\cdot c\cdot d) + (a\cdot b\cdot c\cdot\overline{d}) + (a\cdot b\cdot\overline{c}\cdot d) + \ldots$$

The particular form of representation produced by the above method is called **disjunctive normal form**. It's always a disjunction, whose arguments are conjunctions, whose arguments are either the input Boolean variables or their negations.

NAND alone would suffice

We have described how to describe an *arbitrary Boolean function* with a formula using three operators: NOT, AND, OR. Now we'll show that all these operations can be represented just by NAND, of course, applied multiple times. This is the way to do it:

$$
\begin{aligned}
\text{NOT } a &= \text{NOT}\,(a \text{ AND } a) = a \text{ NAND } a,\\
a \text{ AND } b &= \text{NOT}\,(a \text{ NAND } b) = (a \text{ NAND } b) \text{ NAND}\,(a \text{ NAND } b),\\
a \text{ OR } b &= \text{NOT}\,((\text{NOT } a) \text{ AND}\,(\text{NOT } b)) = (\text{NOT } a) \text{ NAND}\,(\text{NOT } b) =\\
&= (a \text{ NAND } a) \text{ NAND}\,(b \text{ NAND } b).
\end{aligned}
$$

So, there is a single logical operator which suffices to express all Boolean functions. That fact has some practical implications because (as we'll discuss in lecture "Integrated circuits") it allows to simplify the design of integrated circuits.

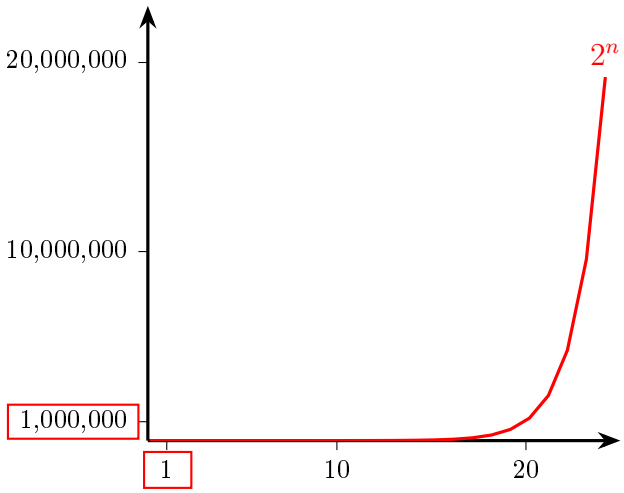
Can this be ever true?

Often, we face a problem in the opposite direction: given a Boolean function (represented by a formula), find a valuation of its arguments for which that function would return value 1. In other words, find which input variables should be true and which should be false in order to make the whole expression true. Of course, that could be solved by computing the whole truth table. However, that requires some reflection on efficiency.

For one input variable, there are two possible valuations (either true or false). For two variables, there are 4 valuations (both true, or just first one, or just second one, or neither). For three variables, we have 8 valuations, then we get 16, 32, 64 etc. That is, the time needed for computations (also called *computational complexity* of the program) is *exponential* in terms of the number of input variables.

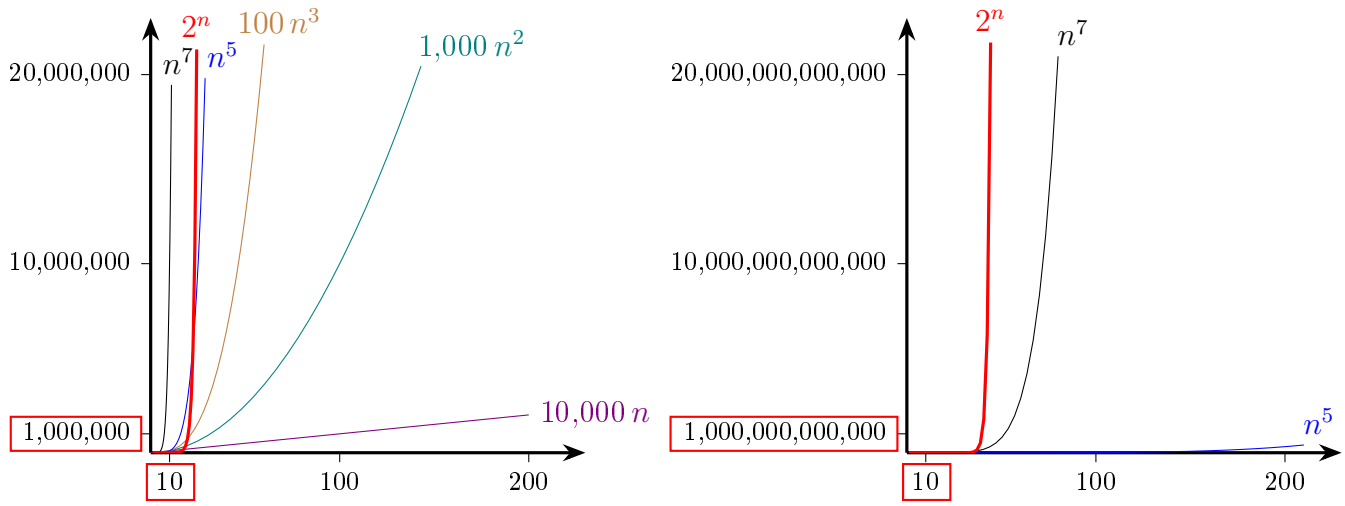
Why we dislike the exponential function

An **exponential function** is a function of the form $f(n) = A^n$, where A is some constant. In the above considerations, we're dealing with an exponential function with $A = 2$. That function is presented on the graph below (caution: the Y axis has drastically different scale than the X axis):



Takeaway: the number of combinations which must be verified grows *very fast*. To realize how fast, let's assume that checking one valuation takes 1 ms (one millisecond); then, for 22 input variables, the whole computation would take over 1 hour; for 39 variables — over 10 years; for 69 variables — more than the age of universe (since the Big Bang)!

When we notice that we're up to performing an exponential amount of computation, it's always worth to think if we couldn't solve the same problem faster. Typically, the goal would be to make the necessary number of operations less than $B \cdot n^A$, where n is the number of input data (in our case, the number of input variables), and A, B are some constants (of course, the smaller, the better). Such complexity is called *polynomial*. The difference between the exponential and polynomial growth rates is shown by the following graphs (again, watch out for the scale on the Y axis):



The graph on the left shows that (for sufficiently large values of n, starting from just about 15) the value of 2^n greatly exceeds not only common polynomials like n^2 or n^3, but even their multiplicities by constants like 100 or 1,000. Moreover, the situation does not change significantly even after increasing the power exponent. Even if, looking on the graph on the left, it might seem that 2^n "loses the race" against n^7, this turns out just to be a matter of scale, as can be clearly seen on the graph on the right. In general, for *every* exponent A and multiplier B, taking sufficiently large n, we will obtain $2^n > B \cdot n^A$. That's why the *theory of complexity* says that "exponential functions grow faster than all polynomial functions".

Finding the truth is hard

After this long digression, we can come back to our problem of finding a valuation for which the given Boolean function returns true. Now, we have bad news. It's a problem which attracted a lot of scientific attention; yet, we still do not know if there exists an algorithm solving that problem in polynomial time.[1] Although these remarks are not constructive, it's good to be aware of them, as it's always better to know upfront what can cause efficiency troubles in programming.

[1] For those interested: Just deciding, given a Boolean function, *whether* there *exists* a valuation leading to a true result, known in computer science as the *satisfiability problem*, belongs to the class of *NP-complete* problems, for which we know no way of solving in polynomial time.