software
development
academy

Introduction to Computer Science

Handbook with exercises

Software Development Academy

march 2019

Table of Contents

Introduction . . . 1

Basics of Computer Science . . . 2

- History of computers . . . 2
- Hardware . . . 3
- Operating system . . . 3
- Computer Networks . . . 3
- Information units . . . 4
- Unix time . . . 4

Numeral systems . . . 6

- Roman Numerals . . . 6
- Hindu-Arabic numeral system . . . 7
- Standard positional numeral systems . . . 7
- Binary system . . . 8
- Hexadecimal system . . . 8
- Unary system . . . 8
- Fractions . . . 9
- Converting decimal numbers to other numeral systems . . . 9

Computers and numbers . . . 12

- Binary Coded Decimal . . . 12
- Usage of the binary numeral system . . . 13
- Signed magnitude representation . . . 13
- Two’s complement . . . 13
- Floating point . . . 14

Logical operators . . . 16

- Logical operators - briefing . . . 16
- The “and” operator . . . 16
- The “or” operator . . . 17
- The “not” operator (negation) . . . 17
- De Morgan’s laws . . . 18

Algorithms . . . 20

- Introduction . . . 20
- Recursiveness . . . 20
- Flowcharts . . . 21
- Pseudocode . . . 24

Introduction

You receive this document as a result of making a decision of taking part in programming course. For this course to be efficient and that covered topic are properly understood you need to have very basic knowledge from Computer Science. In this document there are informations that will help you feel comfortable during the course.

It is very important to read it carefully. Even if you are familiar with some information please read this from cover to cover. On the other hand if some part of material seems hard, you can always go back to it after some time not to skip any topic. In some cases descriptions are simplified so document is consistent and not very complicated to comprehend.

I really encourage to solve exercises, it is for sure the best way of learning how to program. It will look very similar in case of the course - you will get familiar with many terms, but only practical solving of those will guarantee your success.

Basics of Computer Science

History of computers

Since we are about to learn how to program, let's take a while to get familiar with the history of computers. We will write down a few (subjectively chosen) milestones.

- Over 2000 years ago abacus was discovered in Mesopotamia. It is a simple calculating machine but it was used all over Europe and Asia, until adaptation of the Hindu–Arabic numeral system
- In 1623 Wilhelm Schickard built the first mechanical calculator. Well he almost did, cause it had plenty of flaws (it could be damaged by correct usage!)
- In 1642 Blase Pascal (one of the greatest minds of all time!) managed to create a first truly working calculating machine. His father was a tax collector, so he decided to make his life easier
- During the World War I cryptography started to evolve faster and faster. Probably most famous cryptographic device of all time is the Enigma, created in 1919 and modified in later years. It used many mechanical parts as well as the electricity
- During World War II far more complex machines were created. In years 1943 - 1945 United States Army created ENIAC. Computations weren't made mechanically anymore which lead to a huge growth of speed. For many years ENIAC was believed to be the first computer, until Germans and British revealed that they created computers during WWII as well. Another candidate for the first world computer is Atanasoff-Berry Computer created in 1939 by John Atanasoff and Clifford Berry
- In 1954 IBM created first mass-produced computer called 650. It's magnetic data-storage was spinning at 12500 rotations per minute!
- In 1967 Advanced Research Projects Agency created a network (that implemented TCP/IP protocol) called ARPANET. Later that network evolved into the Internet
- In 1968 scientists and engineers at MIT designed Apollo Guidance Computer. They managed to shrink huge computers (much much bigger than refrigerators) to a compact size weighting only 30 kg
- In 1976 Apple-1 was craeted by Steve Wozniak and distributed by Steve Jobs. Their first computer was sold only in 200 copies, but their next one - Apple II was sold in millions
- In later years computers got smaller, faster. They beat human at chess (for a few years even at go!), but all major ideas were developed in the earlier decades. The next big step might be commercial DNA usage for data storage. A truly incredible step would be a powerful commercial quantum computer ("thinking" in many ways at once)
- IBM already offers a quantum computer: IMG Q, but it is still not what we are waiting for - it is based on 5 qubits and does not offer much. That one that we wait for will redefine all of the world's cryptography, because it will be incredibly good at cracking codes

Hardware

Let’s go through basic elements of computer.

Processor

We can understand this component as the one responsible for calculations. Processor receives instructions from computers memory which are then executed with frequency of billions operations per second (in modern computers). Processor often have many cores, which allows to perform many operations in the same time.

Motherboard

It’s a component where you mount other components of computer. We can understand it as an element connecting other components.

Random Access Memory (RAM)

It’s a memory (information storage) which is designed to operate fast. RAM is communicating directly with processor. To make it possible such component is based on complex integrated circuits. This is not persistent memory - after losing power we are losing its content. To quickly go back to previous state after turning on computer operating systems offers so called ‘sleep mode’ which keeps RAM on or ‘hibernation’ where RAM is persisted on disk

Peripherals

Hard disk, Mouse, Monitor, Keyboard

Operating system

After describing hardware let’s talk about software, and more precisely about the most important functions of operating system.

Its responsibility is to give processor time to running tasks. In one moment we can have more than one application running on computer - at least that is our impression. That’s because applications get assigned processor time and upon completion resources are then given to next application (every fraction of second). Some tasks/processes will be more important than others (higher priority) and will use processor time more often.

Similar situation is with RAM. Each application has its assigned memory and when application exists memory is freed.

Very important function is to enable access to peripherals.

Computer Networks

As we know in modern world computer network are essentials. Networks allow different machines to communicate in various ways which we call ‘protocols’. One of such protocols is HTTP which we are using in our web browsers. In input field we type address of resource that we want to access pointing out protocol, remote machine address and concrete resource (http://address/resource). We are then sending request to access resource and we get response with document which is rendered by our browser.

Information exchange through computer networks can be used for many different purposes, f.e. delegating tasks for other machine (can we use computational power of other physical machine)

Information units

To storage information computers are using 'bits' (binary digit). Bit can have one of two values - `0` or `1`. To call memory in efficient way bits are grouped into groups of eight. Eight bits makes one byte. One bit is referred to as 'b', while byte is 'B'. As one bit contains very little information in practice we will talk about bigger units.

For this purpose we use multipliers: *kilo*, *mega*, *giga*, *tera*, *peta* etc. Given the fact we are based on binary system kilo means * 1024 bytes.

- 1024 KB is 1 MB
- 1024 MB is 1 GB
- and so on.

Discrepancy (1024 instead of 1000 for kilo) is omitted in many times when using those short forms for KB, MB etc. that's why in 1998 there was a proposition to use kilo, mega as in SI, where we multiply by 1000 instead of 1024 and new ones would be called kibi, mebi etc. Unfortunately this idea was not very popular and now is leading to bigger discrepancies when we process more data.

Unix time

Most often used representation of time and day in computers is unix time. It is number of seconds that passed since beginning of 1970 in UTC time zone. This way of representing date and time makes operating on date and time with different time zones easier.

Interesting fact is that for many systems maximum value for unix time is 2 147 483 647. This value stands for 3:14:07 19.01.2038 UTC time. To solve this problem number of bits was doubled for representing unix time. Now we will have similar problem in 300 billion years :).

What we have learnt about basics of computer science

- ☑ What are basic computer components
- ☑ What are main functions of operating system
- ☑ How computer networks work
- ☑ What are basic information units
- ☑ What is Unix time

Numeral systems

Roman Numerals

For centuries people used many numeral systems to represent a number. Before we take a closer look at the system that we use nowadays, let's focus on the Roman numeral system, which is still in use (e.g. date and time representation). Roman numerals consist of seven symbols.

Roman Numerals

I = `1`
V = `5`
X = `10`
L = `50`
C = `100`
D = `500`
M = `1000`

To evaluate a number we add symbol values, taking into consideration that if lower-valued symbols are on the left of a symbol with a higher value, we take the higher value and subtract the lower ones.

XI = `10+1`

IX = `10-1`

XLIX = **XL IX** = (50-10) + (10-1) = 40 + 9 = 49

MCMLIX = **M CM L IX** = 1000 + (1000-100) + 50 + (10-1) = 1959

Exercise

1. Evaluate
   a. XVII
   b. XIV
   c. XLIX
   d. CCMMXXC
   e. MDCCCLXXX
2. Find roman numeral representation of numbers
   a. 8
   b. 15
   c. 21
   d. 59

e. 164
f. 1789

Hindu-Arabic numeral system

Currently most commonly used system to represent numbers is Hindu–Arabic numeral system, developed in eighth century by Muhammad ibn Musa al-Khwarizmi (words "Algebra" and "Algorithm" are derived from his name). That system is based on consecutive powers of ten and it requires usage of ten symbols (digits). Those digits are:

0, 1, 2, 3, 4, 5, 6, 7, 8, 9.

We'll pick a number and we will find out, how it's value is evaluated:

$$\mathbf{1958} = \mathbf{1}\times 10^3 + \mathbf{9}\times 10^2 + \mathbf{5}\times 10^1 + \mathbf{8}\times 10^0 = 1000 + 900 + 50 + 8$$

We can clearly see, that each digit tells us, how many specific powers of ten we include in the value.

First number to the right (8 in the example) is responsible for the number of ten to the power of zero (every number to the power of 0 is equal 1). So the first digit to the right says how many units are being added to the value. The next digit corresponds to the number of 10 to the power of 1 (so it's the number of tens). The next one says how many hundreds are added to the value, and so on.

Exercise

Separate number's value to powers of ten (just like we did with the 1958):

a. 4
b. 61
c. 704
d. 26534

Standard positional numeral systems

Based on this system we can develop new ones. Hindu–Arabic numeral system is based on the powers of 10, so from now on let's call it base-10 numeral system (or in short: decimal system).

Let's change the number 10 to something else - for example let's take 3. We needed 10 digits for the decimal system and we will need 3 for the base-3 numeral system (0, 1, 2).

For systems other then base-10 we will use a lower index indicating system base - so if we write a base-3 number, we'll add a 3 number as the lower index next to the number.

$$\mathbf{1021}_3 = \mathbf{1}\times 3^3 + \mathbf{0}\times 3^2 + \mathbf{2}\times 3^1 + \mathbf{1}\times 3^0 = \mathbf{1}\times 27 + \mathbf{0}\times 9 + \mathbf{2}\times 3 + \mathbf{1}\times 1 = 27 + 6 + 1 = 34$$

We see that the value is evaluated similarly to the base-10 number. Again each digit is responsible for specific power of the base (3 in this case).

Let’s try evaluating value of other base numbers.

$$\mathbf{4022}_5 = \mathbf{4}\times5^3 + \mathbf{0}\times5^2 + \mathbf{2}\times5^1 + \mathbf{2}\times5^0 = \mathbf{4}\times125 + \mathbf{0}\times25 + \mathbf{2}\times5 + \mathbf{2}\times1 = 500 + 0 + 10 + 2 = 512$$

$$\mathbf{67015}_9 = \mathbf{6}\times9^4 + \mathbf{7}\times9^3 + \mathbf{0}\times9^2 + \mathbf{1}\times9^1 + \mathbf{5}\times9^0 = \mathbf{6}\times6561 + \mathbf{7}\times729 + \mathbf{0}\times81 + \mathbf{1}\times9 + \mathbf{5}\times1$$
$$= 39366 + 5103 + 9 + 1 = 44479$$

IT heavily uses numeral systems based on powers of 2 (2, 8 [$=2^3$], 16 [$=2^4$]). They are easy to write down using bits, which can only have one of two values.

Binary system

A base-2 number (a binary number) is represented with the usage of only 2 digits (0, 1). For example:

$$\mathbf{1000}_2 = \mathbf{1}\times2^3 + \mathbf{0}\times2^2 + \mathbf{0}\times2^1 + \mathbf{0}\times2^0 = \mathbf{1}\times8 + \mathbf{0}\times4 + \mathbf{0}\times2 + \mathbf{0}\times1 = 8 + 0 + 0 + 0 = 8$$

$$\mathbf{1001011}_2 = \mathbf{1}\times2^6 + \mathbf{0}\times2^5 + \mathbf{0}\times2^4 + \mathbf{1}\times2^3 + \mathbf{0}\times2^2 + \mathbf{1}\times2^1 + \mathbf{1}\times2^0$$
$$= \mathbf{1}\times64 + \mathbf{0}\times32 + \mathbf{0}\times16 + \mathbf{1}\times8 + \mathbf{0}\times4 + \mathbf{1}\times2 + \mathbf{1}\times1 = 64 + 8 + 2 + 1 = 75$$

Hexadecimal system

What about base-16 number (hexadecimal number)? We only have 10 digits, right? Well not exactly - we can have as many digits as we want. We need 16 digits for a number written using hexadecimal numeral system, so let’s choose some 16 symbols, for example:

0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A (10), B (11), C (12), D (13), E (14), F (15).

Numbers in parentheses represent the digit value in base-10 system. Obviously we could choose any characters but these are commonly used in IT world. Now that we have our digits, we’ll evaluate some hexadecimal numbers:

$$\mathbf{1234}_{16} = \mathbf{1}\times16^3 + \mathbf{2}\times16^2 + \mathbf{3}\times16^1 + \mathbf{4}\times16^0 = \mathbf{1}\times4096 + \mathbf{2}\times256 + \mathbf{3}\times16 + \mathbf{4}\times1$$

$$\mathbf{B5C}_{16} = \mathbf{B}\times16^2 + \mathbf{5}\times16^1 + \mathbf{C}\times16^0 = \mathbf{11}\times256 + \mathbf{5}\times16 + \mathbf{12}\times1 = 2816 + 80 + 12 = 2908$$

Unary system

What about base-1 numeral system? It is possible to right a number using only one digit and do the previously described rules (consecutive powers) still apply? Yes, at least if it is an integer value:

$$\mathbf{1111}_1 = \mathbf{1}\times1^3 + \mathbf{1}\times1^2 + \mathbf{1}\times1^1 + \mathbf{1}\times1^0 = 1\times1 + 1\times1 + 1\times1 + 1\times1 = 4$$

We call that the unary system and it its the most simple of all - value is equal to how many times digit 1 appears in the representation.

Exercise

Evaluate value of other base numbers:

a. 31_3
b. 44_5
c. 101_2
d. 10111100_2
e. AB_{16}
f. $51AC_{16}$

Fractions

Let’s expand our thinking into fractions. We will try to use our understanding of decimal system for these fractional numbers:

$$\mathbf{23{,}81}_{10} = \mathbf{2}\times10^1 + \mathbf{3}\times10^0 + \mathbf{8}\times10^{(-1)} + \mathbf{1}\times10^{(-2)} = \mathbf{2}\times10 + \mathbf{3}\times1 + \mathbf{8}\times0{,}1 + \mathbf{1}\times0{,}01 = 20 + 3 + 0{,}8 + 0{,}01$$

We can clearly see that the same rules apply.

Now let’s try it for other bases.

$$\mathbf{2{,}1}_3 = \mathbf{2}\times3^0 + \mathbf{1}\times3^{(-1)} = \mathbf{2}\times1 + \mathbf{1}\times0{,}(3) = 2{,}(3)$$

$$\mathbf{0{,}101}_2 = \mathbf{0}\times2^0 + \mathbf{1}\times2^{(-1)} + \mathbf{0}\times2^{(-2)} + \mathbf{1}\times2^{(-3)} = \mathbf{0}\times1 + \mathbf{1}\times0{,}5 + \mathbf{0}\times0{,}25 + \mathbf{1}\times0{,}125 = 0{,}625$$

Exercise

Find decimal representation of:

a. $1{,}2_4$
b. $1{,}0011_2$
c. $BD{,}3A0C_{16}$

Converting decimal numbers to other numeral systems

One question remains - how can we go the opposite way? Let’s say we have 1453 number and we want it’s representation using base-3 decimal system.

The easiest way would be using division and remainders (the remainder is for example when we have 17 cakes and we want to divide it to 3 children everyone will get 5 cakes and 2 will remain – it is the remainder):

1453÷3 = 484, *reminder=**1***
484÷3 = 161, *reminder=**1***
161÷3 = 53, *reminder=**2***
53÷3 = 17, *reminder=**2***
17÷3 = 5, *reminder=**2***
5÷3 = 1, *reminder=**2***
1÷3 = 0, *reminder=**1***

In each step we take division result, divide it by our base and write down the remainder.

Now let’s write down all the remainders from bottom to the top:

1222211 - it’s the base-3 representation of 1453!

So:

$$1453 = 1222211_3$$

Exercise

Find other based representations of these decimal numbers:

a. 8, base 2
b. 16, base 2
c. 7, base 2
d. 41, base 3
e. 516, base 5
f. 5234, base 16

What have we learned about numeral systems

- there are various ways to right down a number
- how roman numeral system works
- most commonly used numeral system is Hindu–Arabic numeral system (which can also be called decimal numeral system)
- we can use any positive integer as a base for a numeral system
- how we evaluate a representation of a number using based numeral systems
- how we evaluate a fractional number represented using based numeral system
- how can we get representation of a decimal number using other based numeral system

Computers and numbers

Binary Coded Decimal

We already know how to write down numbers using multiple numeral systems. Now let's take a look, how computers do that.

The most important thing to take into consideration is that computers (and electronic devices) store data using bits - an information unit, which can have one of two values: `0` or `1`. We will talk about how computers use bits to represent numbers.

One of the simplest ways to do that is to use BCD (Binary Coded Decimal). It's easy to use, but is a little bit inefficient. We'll assign 4 bits to each digit:

Binary Coded Decimal

0	`0000`
1	`0001`
2	`0010`
3	`0011`
4	`0100`
5	`0101`
6	`0110`
7	`0111`
8	`1000`
9	`1001`

Now, using this conversion table, we can convert a decimal number to a BCD number:

$$\mathbf{514} = 0101\ 0001\ 0100_{\text{BCD}}$$

We group the bits in fours so that the number is easier to read.

We see how simple it is to convert it. But we can also see, that some bits combinations are not used at all (`1010`, `1011`, `1100`, `1101`, `1110`, `1111`). This results in inefficiency, but if there are not many digits to store it's pretty useful.

There are many variants of how can you store digits using BCD. The one that we described is called 8421, because if first bit is 1 we add 8 to the number, if the second one is 1 we add 4, if the third is one we add 2 and if the last bit is equal to 1 we add 1 to our number.

There is one more thing to note - computers store bits in eights - 8 bits is 1 byte. We said that we write a digit using 4 bits, so what now? Two solutions are in use:

- either first four bits of byte are useless (and the last four are the only one that count) and we can store 1 digit in one byte
- or we use 1 byte to store two digits (it's called packed BCD).

Usage of the binary numeral system

Another way two represent numbers is to use binary numeral system. We already learned that this system has only two digits (this aligns perfectly with bits). Also we know how to convert a number representation from decimal numeral system to binary. We'd like to write 57 in binary:

$$57 = 111001_2$$

We will remind ourselves that the underscript means it's written in binary numeral system.

Let's say we want 2 bytes to store a number. Two bytes is 16 bits (one byte is consists of 8 bits), so we can write 57 in binary as:

`0000 0000 0011 1001`

What would be the biggest number we can write using 2 bytes? If each digit means that certain power of two is added to the number, then obviously the biggest number would be:

$$\texttt{1111 1111 1111 1111}_2 = 2^{15} + 2^{14} + \ldots + 2^1 + 2^0 = 65535$$

The lowest number of course is:

$$\texttt{0000 0000 0000 0000}_2 = 0$$

In range 0 - 65535 we have 65536 different numbers, so using two bytes we can store 65536 different values.

Signed magnitude representation

OK, it works for positive integers, what about negative ones? The easiest way is to use the first bit to represent the sign. Let's say that if it is equal to 0 it is a positive number, and if it is equal to 1 the number is negative. This way we can store 57 as:

`0000 0000 0011 1001` And -57 as: `1000 0000 0011 1001`

The highest number is now:

$$\texttt{0111 1111 1111 1111}_{S2} = 2^{14} + 2^{13} + (\ldots) + 2^1 + 2^0 = 32767$$

And the lowest one is:

$$\texttt{1111 1111 1111 1111}_{S2} = -1\times(2^{14} + 2^{13} + (\ldots) + 2^1 + 2^0) = -32767$$

We can see, that can represent the number zero in two ways:

$$\texttt{0000 0000 0000 0000}_{S2} = \texttt{1000 0000 0000 0000}_{S2} = 0$$

Two's complement

If we don't want that inefficiency, we can use a system called two's complement. All the bits represent the same powers of two as in binary numeral system, but the first bit (2^15) says if we subtract it from the result. So the number:

$$\texttt{1000 0000 0000 0000}_{U2} = -2^{15} = -32768$$

is the lowest one, and the:

$$\texttt{0111 1111 1111 1111}_{U2} = 2^{14} + 2^{13} + (\dots) + 2^{1} + 2^{0} = 32767$$

is the greatest one.

We note that there is only one zero:

$$\texttt{0000 0000 0000 0000}_{U2} = 0$$

Floating point

What about very big numbers? And what about numbers with "complicated" fractional parts (such as 1/3).

To store this numbers computers use system called floating point. Let’s say we’ve got a number 14500000000000. It can be written as:

$$1{,}45 \times 10000000000000$$

If we call the green part significand and the blue exponent, we can see, that in decimal system we can write a number using these two (the base will always be 10). It works the same way in binary (the base will be 2!). So the value of the number written in binary will again be calculated as:

$$\texttt{significand x } 2^{\texttt{exponent}}$$

This is how computers often store numbers. Let’s say we have 8 bits to represent the exponent (using two’s complement way), 23 bits to represent the fractional part of the significand (there will be 1 added to the fractional part) and 1 bit to represent whole result’s sign. This is called single-precision floating point format. Important things to note are:

- it uses floating point ($\texttt{significand x } 2^{\texttt{exponent}}$) to store the value
- it can store huge numbers - we can set the exponent to be equal 127, then the result is $\texttt{significand x } 2^{127}$, and 2^{127} is equal to: … you can start with 2 on a calculator and then multiply it by two 126 times.
- it can store really small fractional parts - if we set the exponent to be equal -128 then the result is $\texttt{significand x } 2^{(-128)}$. Again - take the calculator, press 1 and divide it by two 128 times.
- it often does not store the exact value - we have got only 23 bits to store the significand (it’s fraction to be precise), so for many numbers the representation is only an approximation (but often a veeeeeeery close one).

What have we learned about how computers store numbers

- ☑ how to write a number using Binary Coded Decimals
- ☑ how to use binary numeral system to store numbers
- ☑ how to store a negative number (signed magnitude representation, two’s complement)
- ☑ how can we store huge or tiny numbers (floating point)

Logical operators

Logical operators - briefing

Let’s take a look at the sentence:

Mary has blue eyes and red hair or 12 fingers

In real life this sentence could be understood in such ways:

- Either Mary has blue eyes and red hair or she has 12 fingers
- Mary has blue eyes and either red hair or 12 fingers

In IT world we have to be precise. From now on we will say that "and" and "or" words are logical operators. Logical, because they operate on expressions that can be true or false; operators cause they give a result - again a logical one, because if we say "something and some other thing" again this will result in true or false.

The “and” operator

We’ll take a closer look at some logical expressions:

One meter is 100 centimeters and Earth’s natural satellite is the Moon.

We can see, that this sentence is true (because both expressions: "One meter is 100 centimeters" and "Earth’s natural satellite is the Moon" are true).

One kilometer is 100 meters and Earth rotates around the Sun.

This expression is not true - one kilometer is not 100 meters, so the whole sentence is false.

One centimeter is 100 millimeters and Sun rotates around the Earth.

Again - this whole sentence is not true - both expressions are false.

We can generalize these sentences by writing `p` and `q` where `p` is the first expression and `q` is the second one. Both `p` and `q` can be true or false. Table containing all of the results of `p and q` would be:

p	*q*	*p* **and** *q*
true	**true**	**true**
true	**false**	**false**
false	**true**	**false**
false	**false**	**false**

So expression `p and q` is true only when both `p` and `q` are true.

The “or” operator

Now let’s focus on the "or" operator. When we say: *I’ll be there at 5pm or 7pm* we mean either 5pm or 7pm, not including the option that we will be there two times 5pm and 7pm. A computer would consider all the three options as true. The logical operator "or" is true if at least one of the expressions is true, so the table containing the results of `p or q` is:

p	*q*	*p* **or** *q*
true	**true**	**true**
true	**false**	**true**
false	**true**	**true**
false	**false**	**false**

The “not” operator (negation)

Another operator is the "not" operator (negation). If the sentence is negated, then if it was true it becomes false, and if it was false it becomes true.

So the values for `not p` are:

p	**not** *p*
true	**false**
false	**true**

What about more complex expressions, for example: `not p and q or s` - how should we treat this? As `not (p and (q or s))`? Or maybe as `(not p) and (q or s)`? Some operators bind stronger than the others, and the order for the three (not, and, or) is:

1. Not
2. And
3. Or

So `not p and q or s` is the same as `((not p) and q) or s`.

Exercise

1. Add parentheses to the expression, so the value of the expression remains the same:
   a. `not p or q and s or q`
   b. `q and not p or s and t`
   c. `q and s or p and not s and q`
2. Create table containing values for the expression:
   a. `not p and q`
   b. `q or not p`

De Morgan’s laws

Let’s try to write a table of values of the expression `not p and not q` (for the calculations to be easier let’s write down the values of `not p` and `not q` as well).

p	*q*	**not** *p*	**not** *q*	**not** *p* **and not** *q*
true	**true**	**false**	**false**	**false**
true	**false**	**false**	**true**	**false**
false	**true**	**true**	**false**	**false**
false	**false**	**true**	**true**	**true**

So `not p and not q` is true only if `p` is false and `q` is false.

Now let’s do that for `not (p or q)` (we will calculate `p or q` as a step).

p	*q*	*p* **or** *q*	**not (***p* **or** *q***)**
true	**true**	**true**	**false**
true	**false**	**true**	**false**
false	**true**	**true**	**false**
false	**false**	**false**	**true**

We can see, that values are the same! For the same `p` and the same `q` value values of `not p and not q` and `not(p or q)` are exactly the same. This rule is one of the De Morgan’s laws:

```
not(p or q) = not p and not q
```

There is another De Morgan’s law:

```
not(p and q) = not p or not q
```

Exercise

Prove that `not(p and q) = not p or not q` by writing table of values for both expressions.

What have we learned about logical operators

- ☑ value of logical expression is true or false
- ☑ `and` operator
- ☑ `or` operator
- ☑ `not` operator
- ☑ how to combine evaluate complex logical expressions
- ☑ how De Morgan’s laws work

Algorithms

Introduction

In elementary school you may have learned how to find the greatest common divisor (GCD) for two numbers. Let’s say we have two numbers: a and b. We will refer to their greatest common divisor as `GCD(a,b)`.

There is a way called Eucledian Algorithm to find it:

- If `a` = `0` or `a` = `b`, than the result is `b`
- Until `a` != `b` take the higher of `a` and `b` and decrease it’s value by subtracting the lower one.

Let’s try it out. We’ll find a GCD for 32 and 24.

We take the higher one, which is 32 and we decrease it by subtracting the lower one - 24. So we have numbers:

32–24, 24 (after subtracting: 16, 24)

We see that this way of solving the problem of finding GCD is precise, unambiguous. This is why we can call it an algorithm.

Exercise

1. Write an algorithm for finding the sum of elements in a numbers set (you’ve got some numbers and you want to get their sum). Be as precise as possible - it has to work for any set of numbers, and to get the sum you don’t want to do anything that you did not describe.
2. Maybe you know some other algorithm? Like solving a maze using right (or left) hand only? Choose one and write it down.

Recursiveness

Let’s write down Eucledian Algorithm again, but in a bit different form:

- If `a` = `0` or `a` = `b`, than the result is `b`
- If `a` > `b`, than the result is `GCD(a - b, b)`
- If `b > a`, than the result is `GCD(a, b - a)`

We see, that an algorithm to find GCD uses itself!

This kind of algorithm is called recursive - the algorithm wants us to use itself to solve the problem. The term "recursive" doesn’t only apply to algorithms.

Many definitions can be recursive - let’s take a look at another example.

Factorial of an integer - let’s refer to that integer as `N` - is the product of all positive integers lower or equal to the `N`.

For example:

$$5! = 1 \times 2 \times 3 \times 4 \times 5$$

or

$$3! = 1 \times 2 \times 3$$

Generally, we can say, that factorial of `N` is:

$$\mathbf{N!} = 1 \times 2 \times 3 \times \ldots \times (N-1) \times N$$

Where $\mathbf{0!} = 1$

Once again we can define it using recursive way:

- If `N` = `0` then `N!` = `1`
- If `N` ≥ `1` then `N! = (N - 1)! x N`

The most crucial part is:

$$N! = (N - 1)! \times N.$$

We see, that the definition of `N!` uses itself once again.

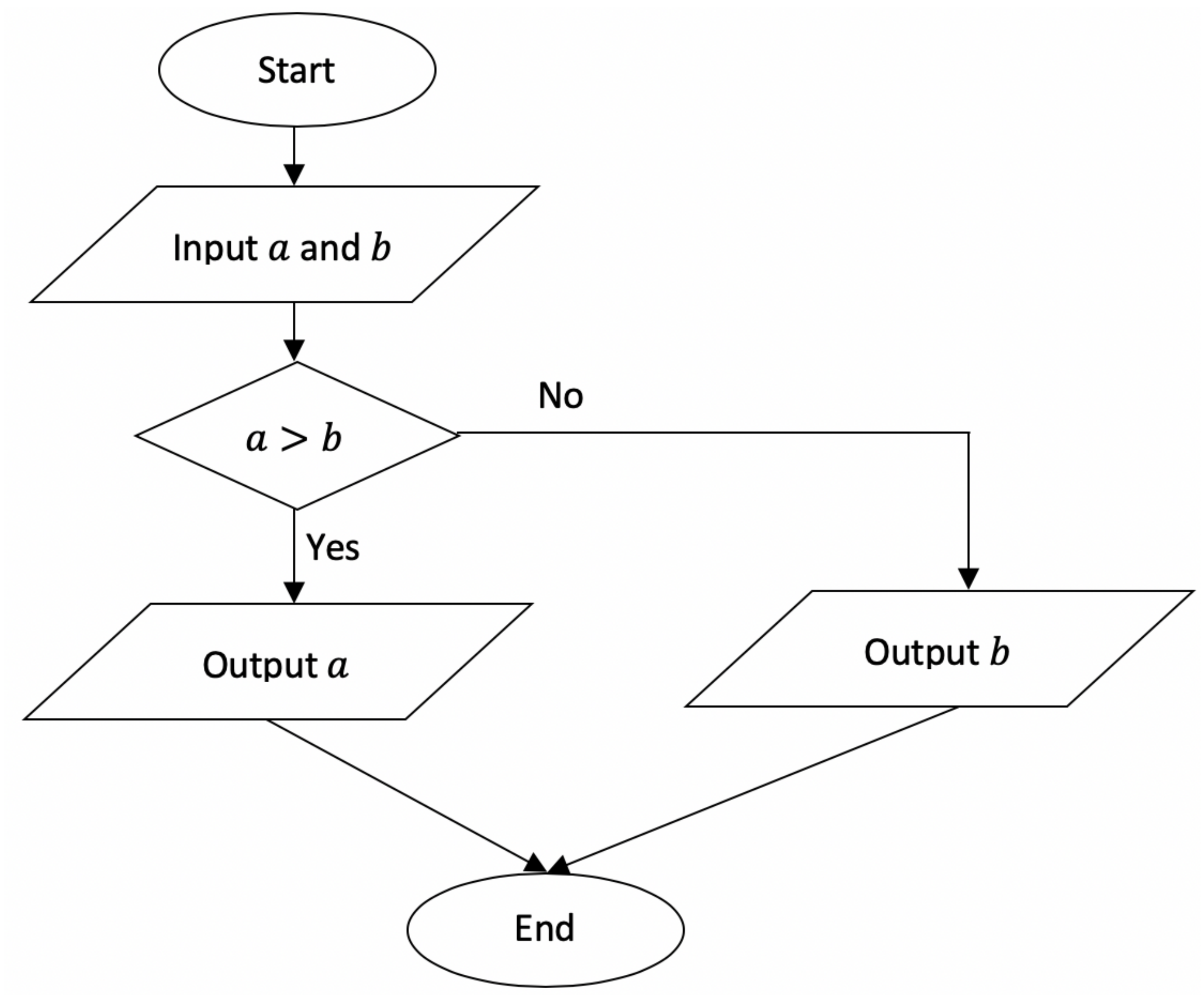
Exercise

We've got a series of elements. You start with the first element. Write down an algorithm for finding the last element in the series using recursiveness.

Flowcharts

Algorithms can be described in many ways. One of them is to use flowcharts - a diagram where parts are equivalents to steps in algorithm.

Let's take a look at a very simple flowchart.



It shows how to find the greater out of two numbers

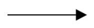
It works as follows:

1. The start.
2. We take input - two numbers
3. We compare the numbers (check if a is greater than b)
4. If a is greater than we output a.
5. If b is greater than we output b.
6. The end.

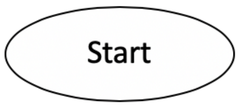
For example we want to use it to find the greater of two numbers: 54 and 25. All of the steps would be:

1. We start our process.
2. We take input - numbers 54 as number a and 25 as number b
3. We compare the numbers (check if 54 is greater than 25)
4. a is greater so we output a (54).
5. The end.

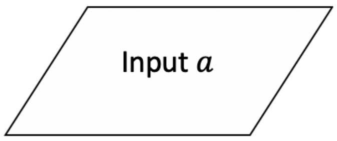
Now that we showed an example let's see some of the symbols that we can put into a flowchart.



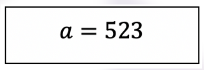
represents the flow order.



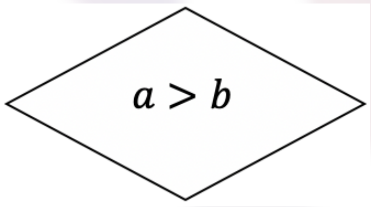
represents starting point of our process.



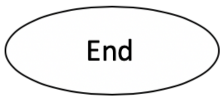
input or output operation - If we want to get data "from the outside" or we want to send some data "to the outside", than we use an input/output operation. We can use it if we want to write the result or if we need input data.



process operation - we use it to perform an assignment operation - if we want to update a value of some variable.

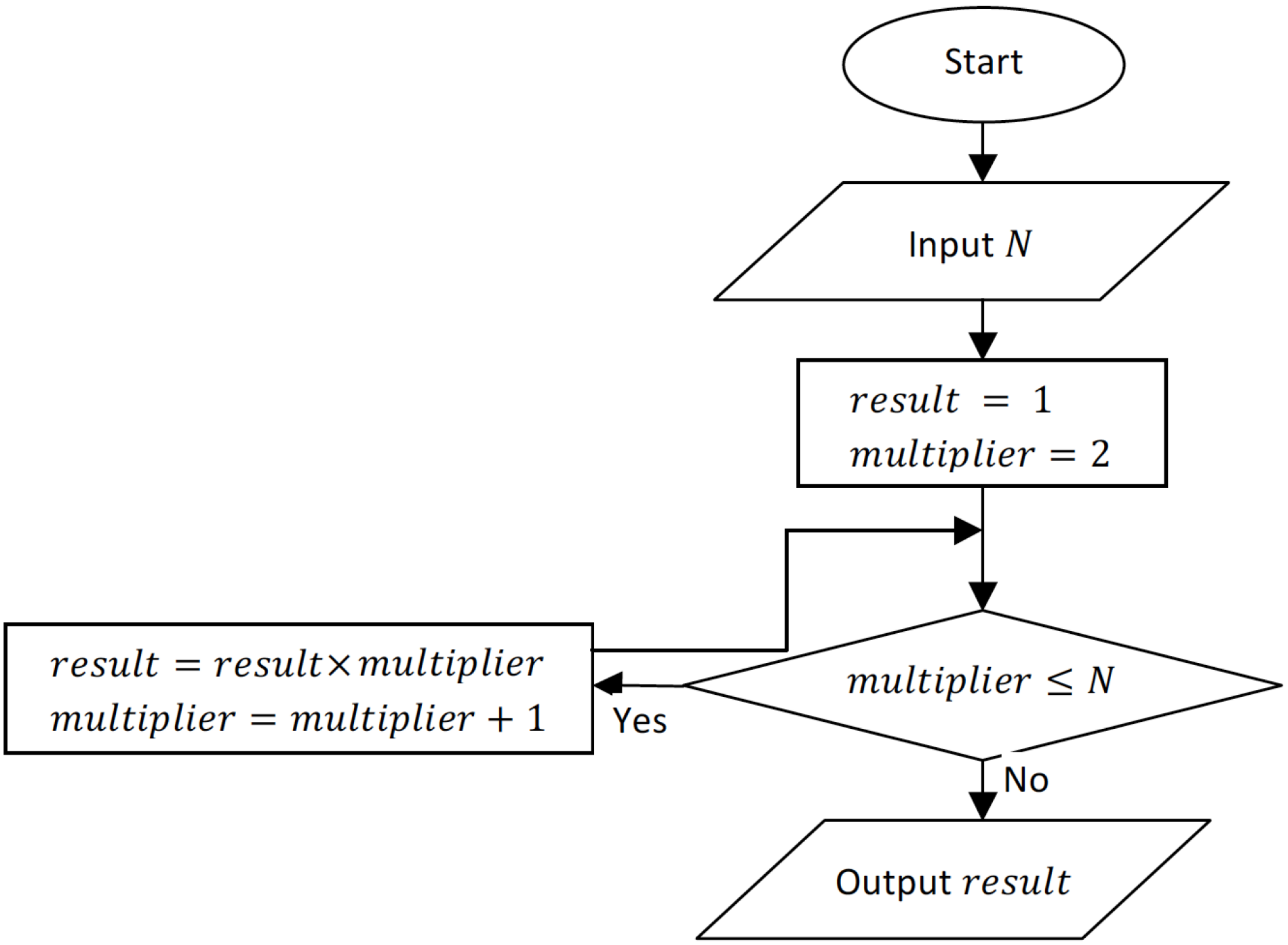


condition - if we want our chart to go in two ways depending on a condition we use the decision symbol. The condition is written inside the symbol and there are two arrows connected to it - the flow depends on whether condition is true.



represents the end of our process.

We can use flowcharts to represent various processes. Let's write down flowchart for finding `N`!.



So we take `N` as an input. Then we declare result to be equal `1` and `multiplier` as `2`. While `multiplier` ≤ `N` we multiply result by current value `multiplier` and increase `multiplier` value by `1`.

Exercise

1. Create a flowchart for a process of finding the greatest out of three numbers.
2. Create a flowchart which represents Eucledian Algorithm.

Pseudocode

Another way to represent processes is pseudocode. Let's take a look at the example:

```
Start
Input a, b
If a > b Then
    output a
Else
    output b
End If
End
```

We can see similarities to the flowchart we've drawn earlier. Again it represents algorithm of finding greater of two numbers. There are many different ways of writing pseudocode, but the most important rule is to be precise!

The process for inputs 12 and 15 would be as follows:

1. Process starts
2. We take our inputs (12 and 15) as `a` and `b`
3. We check if `a` is greather than `b` (which is not true), so we go to the "Else" part
4. We output `b`
5. Process ends

Let's try to show process of finding `N`! using pseudocode:

```
Start
Input N
result = 1
multiplier = 2
While multiplier <= N
    result = result * multiplier
    multiplier = multiplier + 1
End While
output result
End
```

We will use it to find the value of `3`!

1. We start our process
2. The input is `3` (we assign value `3` to the `N`)
3. We declare variable `result` and assign value `1` to it.
4. We declare `multiplier` variable and say that at start it is equal to `2`.
5. While `multiplier` is less or equal to `N` (`N` is `3`, `multiplier` is `2`):

   ```
   result = result * multiplier // in this case: result = 1 * 2
   multiplier = multiplier + 1  // in this case: multiplier = 2 + 1
   ```

6. While `multiplier` is less or equal to `N` (`N` is `3`, `multiplier` is `3`):

   ```
   result = result * multiplier  // in this case: result = 2 * 3
   multiplier = multiplier + 1   // in this case: multiplier = 3 + 1
   ```

7. While `multiplier` is less or equal to `N` (`N` is `3`, `multiplier` is `4`) - condition is not true so the loop is interrupted.
8. Output `result`: `6`
9. End

Exercise

1. Write pseudocode for a process of finding the greatest out of three numbers.
2. Write pseudocode which represents Eucledian Algorithm.

What have we learned about algorithms

- ✓ what algorithm is
- ✓ what is recursiveness
- ✓ how can we use a flowchart to describe a process
- ✓ how can we use pseudocode to describe a process