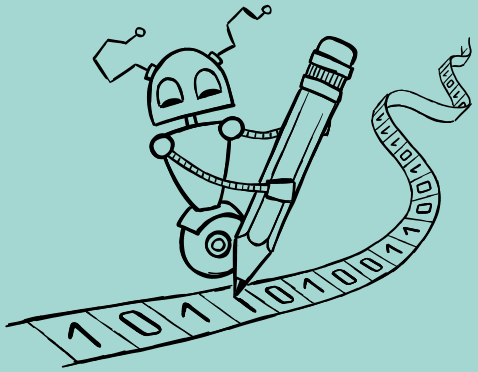
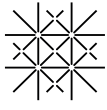


2^5 Things I Learned in Computer Science

University of Basel, Dept of Mathematics and Computer Science

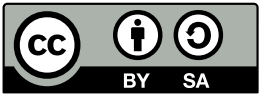
2^5 Things I Learned in Computer Science

Department of Mathematics and Computer Science



University of Basel

© 2019 University of Basel, Department of Mathematics and Computer Science
Spiegelgasse 1, CH – 4051 Basel, Switzerland
https://dmi.unibas.ch/



This work is licensed under a *Creative Commons Attribution-ShareAlike 4.0 International License* (CC BY-SA 4.0).

https://creativecommons.org/licenses/by-sa/4.0/legalcode

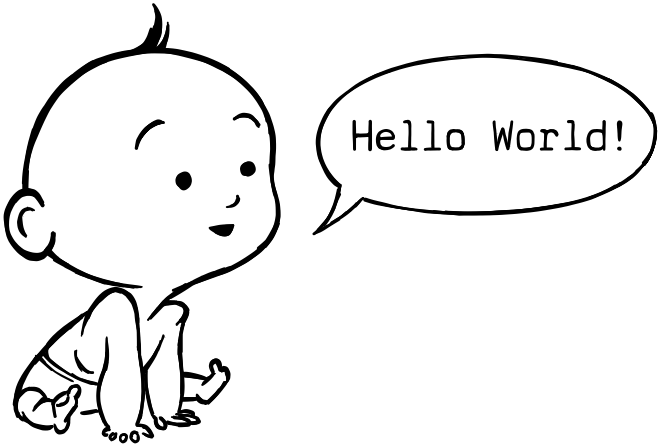
Author's Note

Studying Computer Science is like taking a deep dive into an endless ocean of ideas, concepts, and tools. It is easy to get lost in this wide landscape, even more so if it is not one's main field of expertise.

This is why Computer Science students of the University of Basel set out to identify the essential concepts and ideas they discovered during their studies. The intention was to go beyond the scope of traditional textbooks and also include the pecularities that are not formally part of typical Computer Science curricula. Finally, the goal was to present the fruit of this work to people outside of our field. The project was organized in the form of a Master's level seminar, which had its first iteration during the spring semester 2019.

The result is **2^5 Things I Learned in Computer Science**, a collection of short illustrated articles. It is available in the form of this booklet, and online on the companion website https://tilics.dmi.unibas.ch

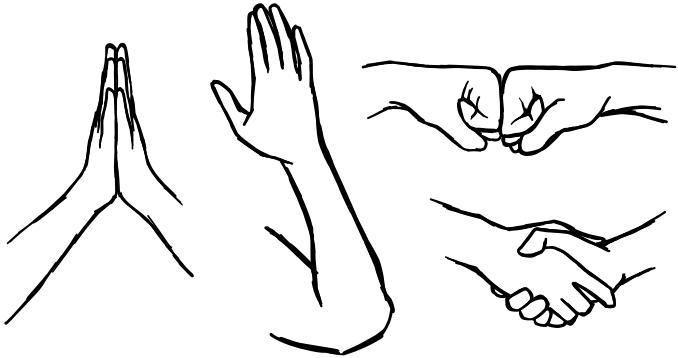
We hope that you enjoy these info bytes,
and that you pass them on to your friends!


Hello World!

0

The first words of a computer scientist

The first piece of code you usually find in a programming book or tutorial is a “Hello World!” program. The purpose of this program is to display the text `Hello World!`. Since it is rather easy to write a program to print text, one can familiarize themselves with the basics of a programming in a certain language. Due to this tradition, these are usually the first lines of code a programmer writes in their life.



1

Introductions are universal

When people meet each other for the first time, there is usually some sort of formal introduction. This fundamental process also applies to computer science and is called a handshake. During a handshake, different processes agree to communicate with each other based on a set of rules, known as a protocol. Sometimes the protocol involves identifying each other using private keys or passwords.

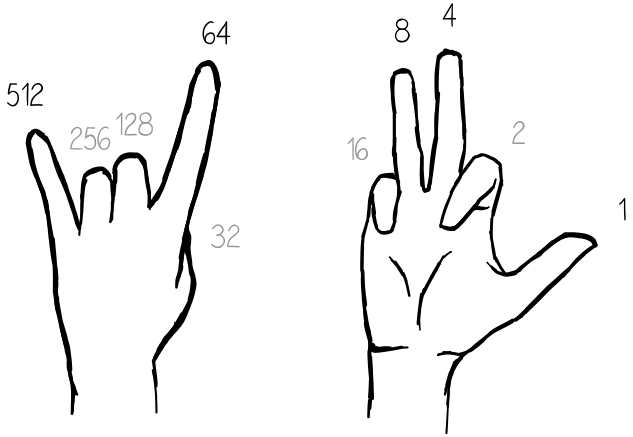
2

Being lazy is a good thing

A lazy person is always searching for ways to make things faster and more efficient.

Instead of doing the same thing over and over again, a good computer scientist tries to automate the task. This not only improves the reliability, but also saves time in the long run. Furthermore, a good computer scientist is also lazy when it comes to writing code. Rather than solving a specific problem, a good programmer develops a generic solution that can be reused for similar problems.

Good programmers are diligent in order to be lazy. Or with other words: They are strategically lazy. And that's a good thing.



512 + 64 + 8 + 4 + 1 = 589

3

Counting to more than ten with two hands

Normally you count with your fingers by stretching them out one after another. You don't care about which ones are stretched out, but only about the amount of them. Each of them is as significant as the other.

By changing the significance in the way that the right thumb has significance 1 and the left neighbor neighbor is always twice as significant we can get 1024 different numbers: `1 + 2 + 4 + 8 + 16 + 32 + 64 + 128 + 256 + 512 = 1023, and 0`. This is called binary counting and exactly the way a computer counts but with zeros and ones instead of bent and stretched fingers.

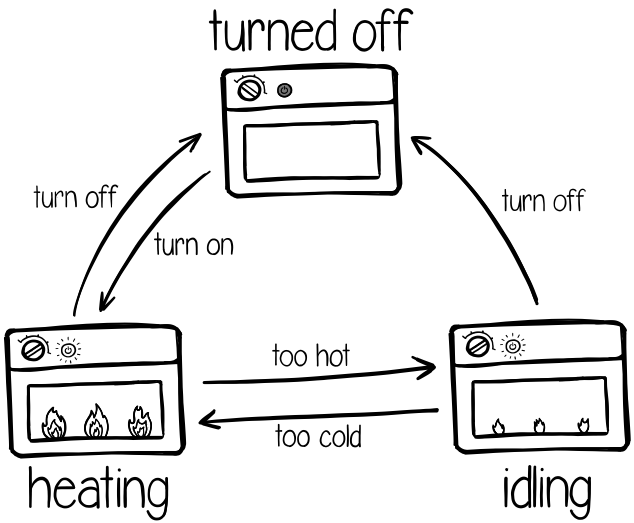

Loading...

Does it scale?

4

Let's say you want to watch a movie. Downloading it would be a matter of minutes. When ordering it online, the shipment might take a week to arrive. Imagine now you want to watch ten movies. Downloading them might take a an hour or two now. But the shipment would still arrive in one week. Now what about twenty movies? Or maybe even a hundred?

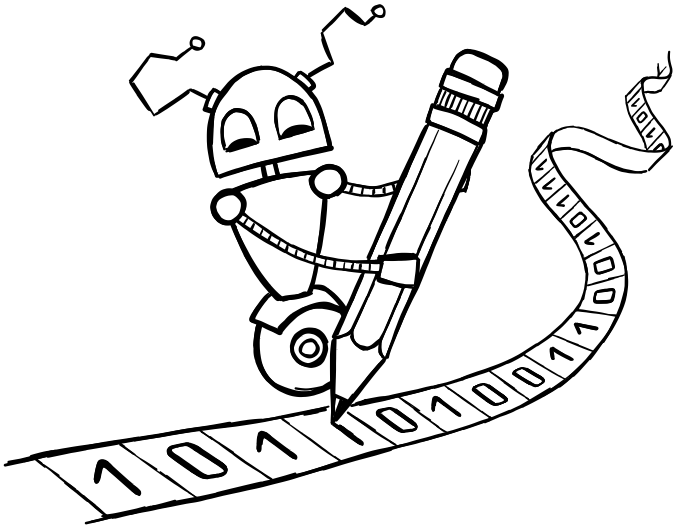
When analyzing the complexity of an algorithm, it is important to think about the *scalability* of a program. It tells you how the runtime of a program depends on the number of elements processed. In the initial example, the time needed to download the movies depends on the amount. Computer scientists would say: The runtime scales with n, where n stands for the number of movies. When having them shipped, the waiting time doesn't depend on this number, which means it doesn't scale.


turned off
turn off
turn on
turn off
too hot
too cold
heating
idling

Think about the different states of your system

State machines are a concise way of describing systems with clearly distinguishable phases or states. The concept of state machines was developed in the context of computer systems but is now also used by engineers from other disciplines.

A state machine is a system which defines its current status through a set of states. The current state determines what the system is doing. According to the diagram, the state is reflected by whether the heating element is activated or not. The transition between the states happens based on events. These events can be triggered by external factors, for example a user pressing a button or a sensor reading, and by internal factors such as the result of a computation.


1 0 1 1 0 1 0 0 1 1 0 1 0 0 1 1 0 1 0 0

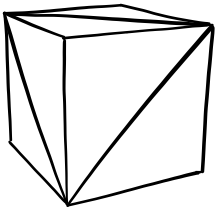
The essence of computation: the Turing machine

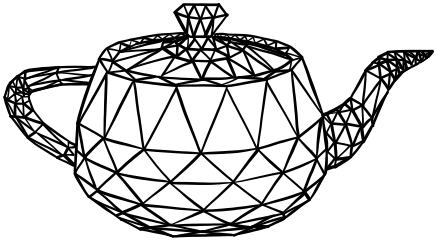
6

The Turing machine is an imaginary model of a machine. It describes what can be computed by following an algorithm. It prints information in a coded form on a tape. This tape is divided into squares. Each of these squares contains either a 0 or a 1.

As a state machine, the Turing machine reads the content of the currently selected square and executes the algorithm associated with its current state. Every possible state of the Turing machine has an algorithm associated with it. Depending on the algorithm, the machine modifies the content of a square and moves the tape to the left or the right.

The Turing machine is a quite simple concept, but it contains the essence of computation: Every problem that can be computed, can theoretically also be computed by a Turing machine. This makes the Turing machine an important concept in theoretical computer science.

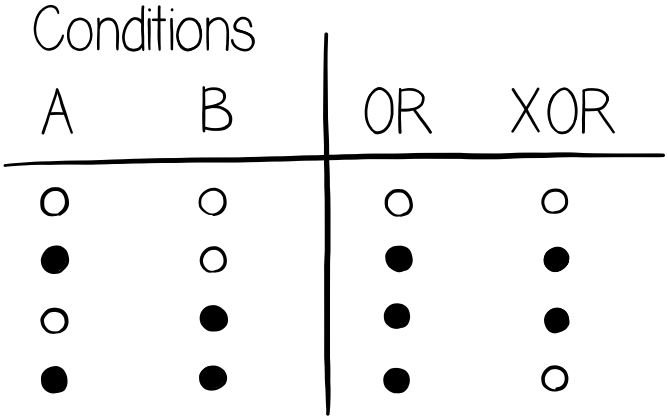




World of triangles

The 3D pictures seen in computer animated films or computer games consist of many small triangles. The shape of a figure in such an image comprises of many points. To visualize a surface, a set of those points is connected by lines. The mathematical figure which emerges through this process is called polygon. The smallest polygon possible is the triangle, where only three points are connected. Furthermore, a triangle guarantees that all the points of the polygon are on the same plane.

The more triangles are used, the more detailed a surface is. Whereas a simple object with flat surfaces like a cube can be modeled by using only a few triangles, a more complex structure like a teapot requires many more triangles for representing its round shape.

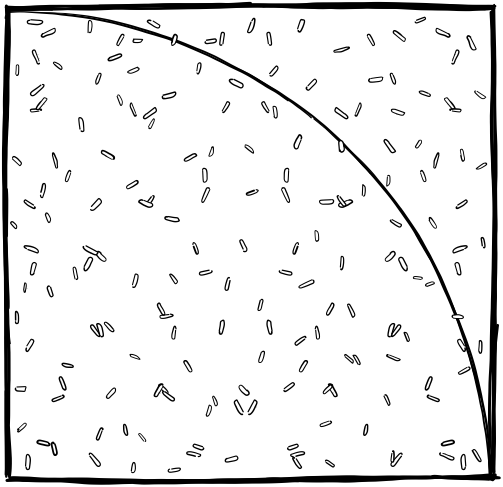

Conditions
A
B
OR
XOR

There are two different kinds of OR

The question whether an “or” can also be interpreted as an “and” is a common source of confusion. While as humans we can often resolve this confusion by looking at the context, computers require a strict definition.

In mathematical logic, the foundation of computer science, it is therefore distinguished between “OR” and “exclusive OR”. The latter is often abbreviated as XOR. In English, we can express an XOR by using an “either . . . or . . . ” construction.

The OR operator is always defined as an “and/or”. It is fulfilled if at least one of its conditions is fulfilled. It is therefore also called “inclusive or”. An “exclusive or” on the other hand is fulfilled if (and only if) exactly one of the conditions is fulfilled. If both of its conditions are met, the “exclusive or” is no longer fulfilled.

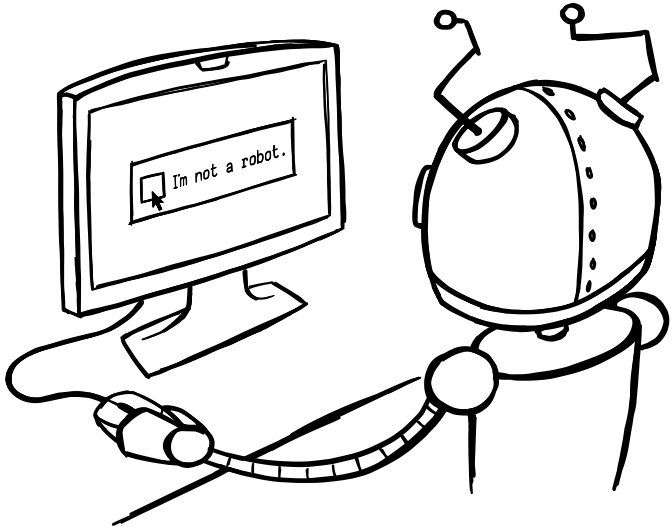


How to calculate π with rice

Draw a square and a quarter circle in it on a paper and throw a hand full of rice on it. Done!

Well almost done. Now you count how many grains of rice landed inside the square and how many landed inside the quarter circle. The ratio of these two numbers approximates the ratio of the two areas ($\pi \times r^2/4$) / r^2 = $\pi/4$. Multiply the ratio by 4 and you get π, well it is an approximation. If you are not happy with the accuracy just use more rice.

This is one example of the Monte Carlo method. The key is to avoid computing a complicated formula by using random samples in a clever way.


I'm not a robot.

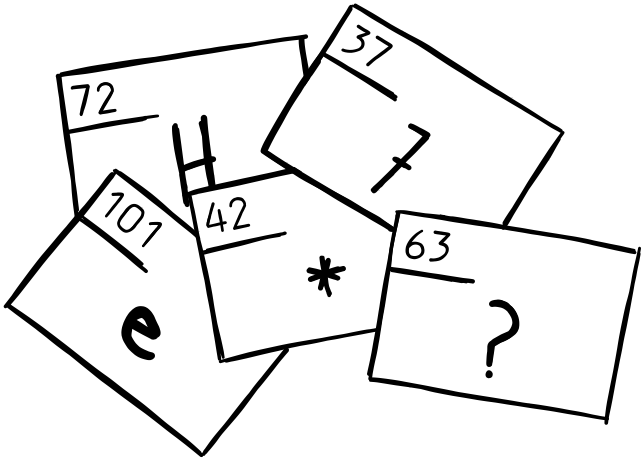
Can a computer talk like a human?

Are you sure it is a human answering your question in the chat support of your favorite online store? If not, how would you find out? What sounds like an easy question turns out to be rather difficult.

10

In 1950, a computer science pioneer named Alan Turing already thought about this problem and presented the Turing test. The test is passed if a human judge interviewing a human and a machine is unable to distinguish between them based on their answers. The first program that managed to convince some of the judges was ELIZA. It managed to mimic a psychotherapist by reflecting the questions back to the questioner. While there have been several promising attempts in the recent years, yet no program managed to fully pass the test.

However, the next time someone in the chat support of your telecom company tries to calm you down–ask yourself: Am I talking with a human?


72
37
7
101
e
42
*
63
?

Every letter is a number

In computers, letters are represented as numbers. To encode them, every letter is assigned its own decimal number between 0 and 127. The “H”, for example, is encoded as the 72. The string `Hello world` is stored as a sequence of numbers:

```
 H  e   l   l   o  space  w   o   r   l   d
72 101 108 108 111   32  119 111 114 108 100
```


As everything is stored in binary in computers, those numbers are translated in binary code. Not only are the capital letters encoded in this way, but also the lowercase letters, the digits `0-9` and punctuation symbols.

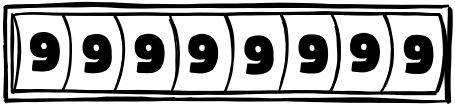
This very common character encoding is called ASCII (American Standard Code for Information Interchange). Today, the international standard for encoding is called Unicode. It is way bigger than ASCII — big enough that every possible character of every language may have its own digital code.

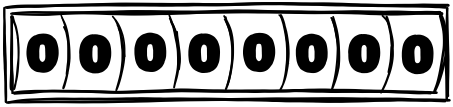

41
42
DAYS WITHOUT
ACCIDENT

Why uptime matters

The time since a computer was started is referred to as its **uptime**. When the computer is rebooted (or crashes), its uptime is reset to zero. The time the computer is not running is called **downtime**.

When running a **service**, such as a website, we want it to be available at all times. This means that the **server** the website is running on must be available all the time. As a high uptime means that a server has been running a long time without interruption, it can indicate that the provided services are also available with few interruptions.


9 9 9 9 9 9 9 9

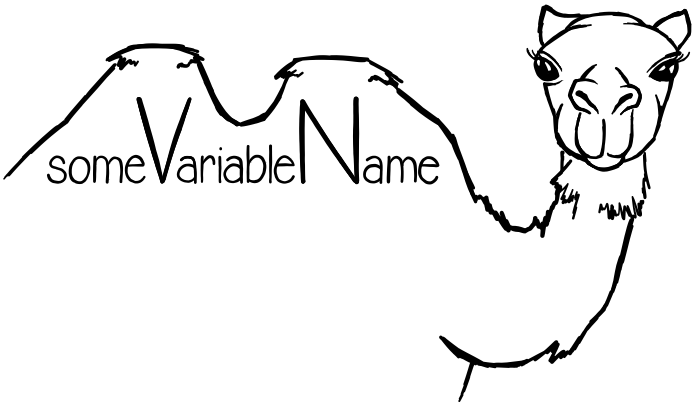

0 0 0 0 0 0 0 0

Overflow

Imagine an odometer of a car. There are eight spaces each of which can display a one-digit number. Therefore, the highest number that can be represented is 99'999'999, after which the odometer turns to 00'000'000 again.

The same thing can happen in a computer. Every number has a fixed amount of bits it can use to store its value. As soon as all these bits are set, increasing the number again will cause an overflow, meaning that the number will wrap around to its smallest value.

This behavior has led to notorious bugs, such as the software of the Ariane 5 rocket malfunctioning due to an unexpected overflow. This caused a crash shortly after takeoff, costing several hundred million dollars.


someVariableName

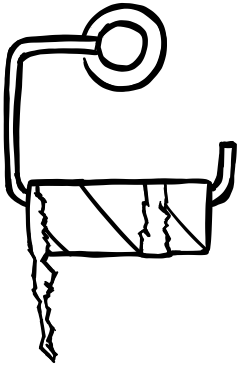
CamelCase, or why coding conventions matter

One problem that often arises when writing code is the naming of variables. Often multi-word names are used for variables, but in most programming languages it is not possible to use spaces in a variable name.

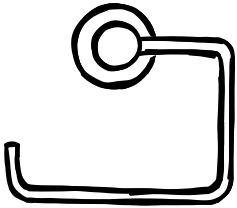
One programming style that tries to structure this is called CamelCase. It is the practice of writing each word in the middle of a phrase with a capital letter:

```
thisGreatlyImprovesReadabilityComparedTo
    whenwewouldusenocapitalizationatall
```

Zero

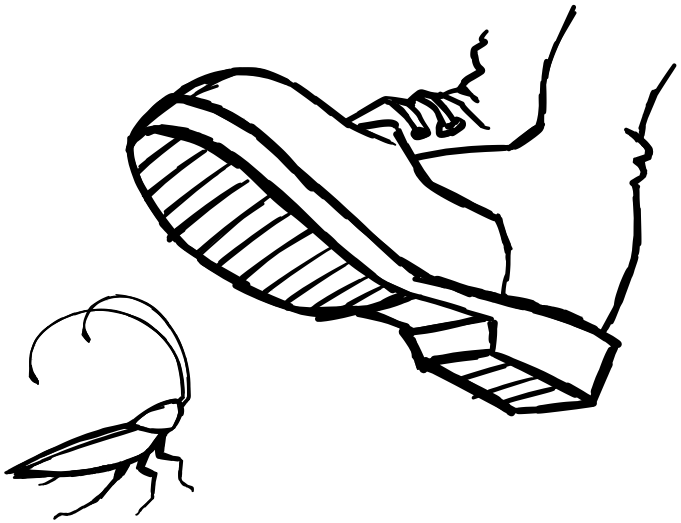


Null



Zero is not null

The null value has a special meaning in computer science. It is used to declare that a value is not yet defined or known. The specialty of null is that although it shows us that a value is missing, it itself is a value. So, it is possible to compare two nonexistent values. Suppose we have data about a patient; it does make a difference whether the patient has no disease (0) or is not yet diagnosed (null).

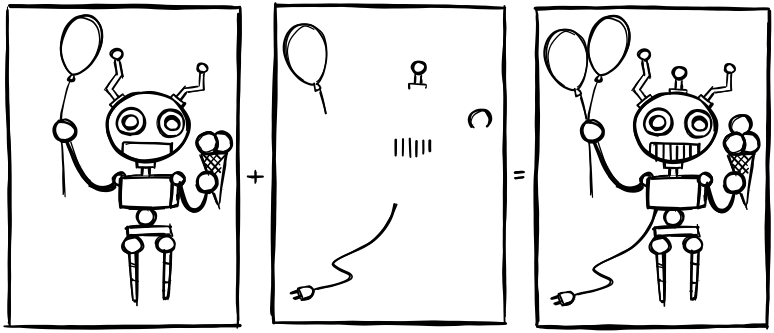


We all hate bugs

Bugs creep into your house without notice, they eat your food and clog your sink. They hide in corners, behind large furniture and are most of the times hard to find. When you get rid of a bug that was bothering you for a long time, you feel relieved.

Unlike a real life bug, a software bug causes unintended behavior which can cause minor errors or even crash your system. It is often very hard to find bugs in software and can sometimes be attributed to one single character that was wrongly placed. The act of finding bugs is called debugging. Like pest control, the code is searched and cleaned from every bug that influences the behavior negatively.

Unfortunately, while fixing software bugs, the programmer accidentally adds other bugs. The cycle is endless.



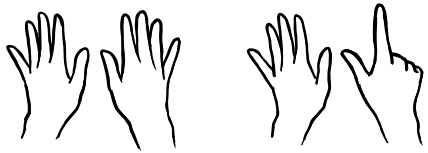
Why patching is like picture puzzles

If you have a corrected version of a buggy program, why send the whole program again instead of just the small differences?

This is how updates for your smartphone work: you will receive the set of differences –a so called *patch*– that your smartphone applies to the old and buggy program.

Now go and find the differences: Did we spot them all?

17_dec

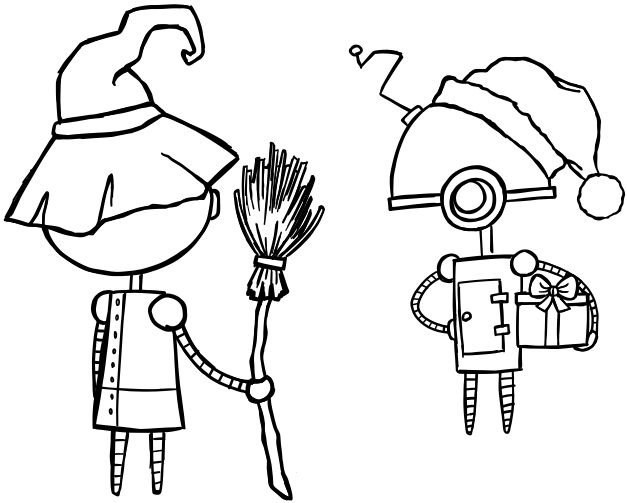


21_oct

What if humans had 8 fingers instead of 10?

Whenever we are dealing with numbers in our daily lives, we use the so called *decimal system*. This means that our entire counting system is based on the number 10. A given number, for example 17, actually means *"1 time the number 10, plus 7 times the number 1"*. It's the same as counting with your fingers. Once you reached 10, you remember the result and start over again with one finger.

Now what if we had 8 fingers instead? Our counting system might be based on the number 8. This would be called *octal system*. Our decimal number 17, also called $\boxed{17_{\texttt{dec}}}$ to prevent confusion, would instead be written as $\boxed{21_{\texttt{oct}}}$, which actually stands for *"2 times the number 8, plus 1 time the number 1"*.



Why does the computer scientist confuse halloween and christmas? Because $\boxed{25_{dec}} = \boxed{31_{oct}}$

A computer stores everything with 1s and 0s. To encode any number as a string of 1s and 0s you use the binary system. It is similar to the decimal system normal humans use, but it has only two digits 1 and 0.

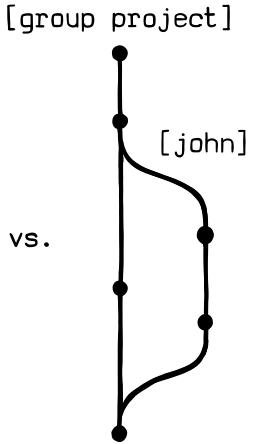
In the decimal system, you have the base 10 and can rewrite the number 25_{dec} as $2*10^1 + 5*10^0$. In the octal system the number 31_{oct} can be represented as $3*8^1 + 1*8^0$, which is the same as 25_{dec}.

There are also other systems like hexadecimal with the 16 digits *0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F* and the sexagesimal system with 60 different digits. To get from one system to the other, you divide by the base with remainder until you reach 0. Then, you string the remainders of these calculations together starting with the one computed last.

group_project

- code.py
- code_new.py
- code_johns_version.py
- code_new_final.py
- code_johns_version2.py
- code_combined.py

vs.



Always use a Version Control System

It's no secret that backing up your data is a good idea, and the same goes for your code. However, keeping a lot of slightly renamed versions of your files for every little modification is probably not the way to go.

Version Control Systems (VCS) allow not only to back up your files, but also, more importantly, help you to keep track of the different versions and what changed in each of them. Even better, multiple people can simultaneously work on one file and then easily combine the different changes.

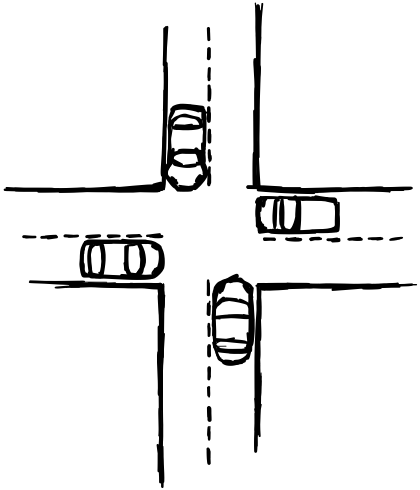
0 1 2 3
4 5 6 7
8 9 A B
C D E R

Why are you hiding this?

The idea for information hiding is to reduce complexity and indirectly interact with the information/data using an interface. But why would we do this?

Suppose we want to get a drink from a vending machine. We only need to select the drink we want and then pay for it. We never need to see the inner workings or the logic behind the vending machine.

This is the same for computer programs. We use interfaces to hide the information/methods. This also means that we can interchange the hidden information easily as long as we do not change the interface. Furthermore, since we do not know exactly how the programs or machines work, we will have a harder time to manipulate them.

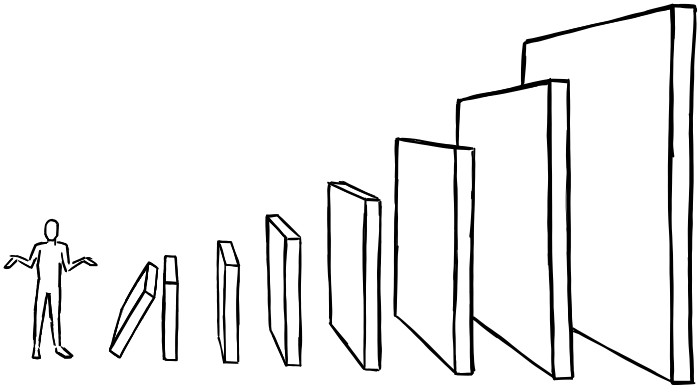


Deadlock

Imagine an uncontrolled intersection without any traffic signs. Here, drivers have to respect the priority to the right system. In the situation shown in the left picture, none of the cars can continue. In practice, one driver would give a hand signal to let one of the other cars go first.

A similar situation can occur when several computer processes running in parallel want to access shared resources. In a *circular wait* scenario, processes block each other from continuing execution as they wait for a resource held by one of their peers.

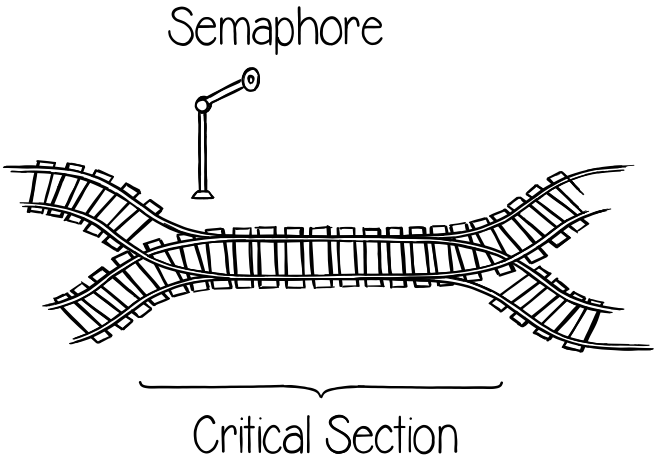
Since computer processes cannot hand wave at each other, clear rules are needed to avoid this situation in the first place.



Errors propagate!

In daily life it is common to neglect small inaccuracies, be it rounding up change or not being exactly on time.

However, the world of computing is less forgiving. Due to the representation of (floating-point) numbers, there nearly always is a little bit of inaccuracy present and careless computations can increase it substantially. Depending on the problem the errors can increase explosively for each step of an algorithm leading to unfortunate results.

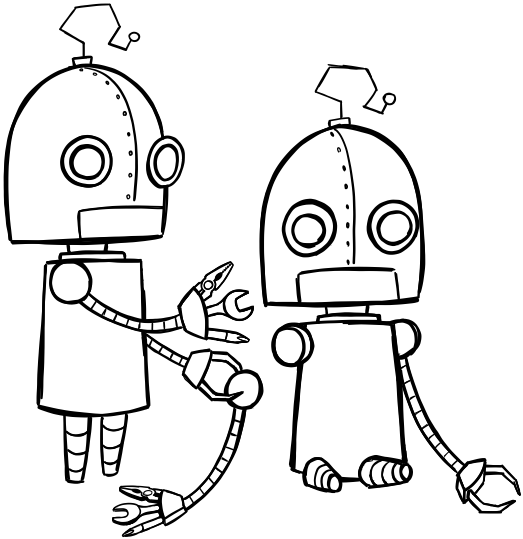

Semaphore
Critical Section

Semaphores prevent crashes

Imagine a railway intersection where only one train can pass at a time. If another train tries to pass the intersection while it's already being used, the trains will crash.

The same thing can happen in a computer. As soon as a resource gets changed by two different sources at the same time, things can go wrong.

In rail traffic, this is solved, by having a *semaphore* to only ever allow one train to pass the intersection. The same is done in computers. Here, a *semaphore* only ever allows one source to access a critical section. Everyone else has to wait, thus preventing crashes or any unintended behavior.



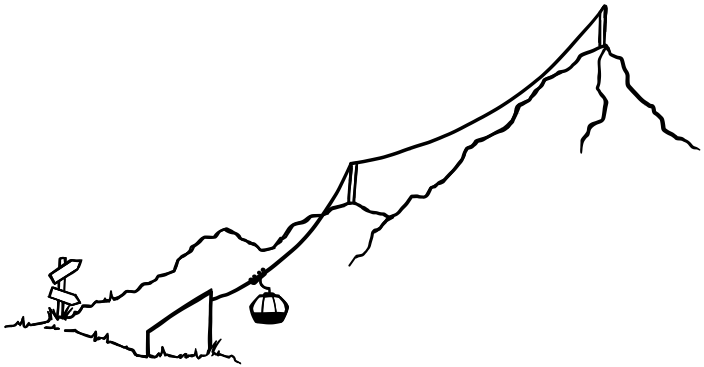
The fixpoint of describing yourself

What could be easier than to write a program that produces its own code as output?

It's not that easy — even a well-versed programmer will need to spend some hours to figure out the trick (which can be different for each programming language).

One trick is that the program must contain a description of itself, inside its own code. Because outputting the self-description also needs code, the self-description must be somehow compressed.

But once written, such a program –also called a "Quine"– reproduces itself each time it is run, and this in turn is called a fixpoint. One could call this the fixpoint of life because any lifeform (including robots) must master this trick if they wanted to build themselves.



Being greedy won‘t make you rich

A very intuitive approach lies at the bottom of so called *greedy algorithms*: at every step of the problem, do what looks best *at that moment*. To imagine this, think about how you hand out change. First, you pick the most valuable coin that covers most of the amount. Then you pick the second largest valued coin of the remaining change. And so on. You‘ll most definitely be done faster than if you just started assembling random coins.

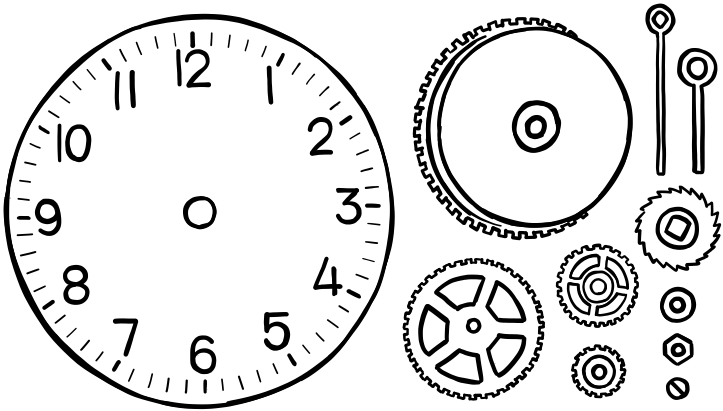
Greedy algorithms unfortunately only work for rather simple problems. Often it’s necessary to plan ahead more than just one step. When climbing a mountain for example, taking at every intersection the path with the steepest slope is a valid approach that will probably bring you to the top. However, by being greedy you can either miss better opportunities that would bring you to the top faster or end up never reaching the true highest point.

Guess fast or wait!

Imagine you're in an unfamiliar city, trying to find a place to stay. You decide to use your phone to navigate. Since you have a good sense of direction, you already start walking, while looking up the way on your phone.

If you guessed the direction correctly, you didn't waste any time waiting for the result, and can continue on your way. If your guess was wrong, you have to turn around.

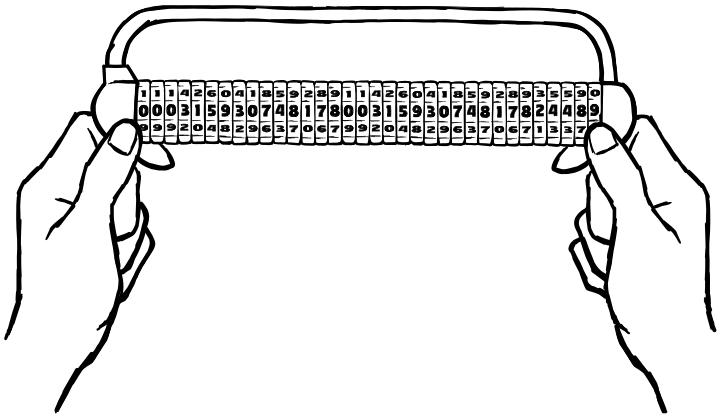
A modern CPU does this as well: When its course of action depends on a result it would have to wait for, it makes an educated guess and continues along this path. This is called **speculative execution**. If the guess was wrong, the CPU has to discard all the actions it has done in error. It did not waste any time though, as it would have spent the time waiting instead. However, if the guess turns out to be correct, the CPU just gained a little speed boost.


12
1
2
3
4
5
6
7
8
9
10
11

Reverse engineering

Reverse engineering describes the process of analyzing an existing apparatus to figure out how it works and how it was made. The same can be done for a computer program, but instead of looking at screws and cogwheels, we are looking at the machine instructions. These are effectively the 0's and 1's that tell the computer what to do. This is obviously much harder to understand than the code that was used to generate the program.

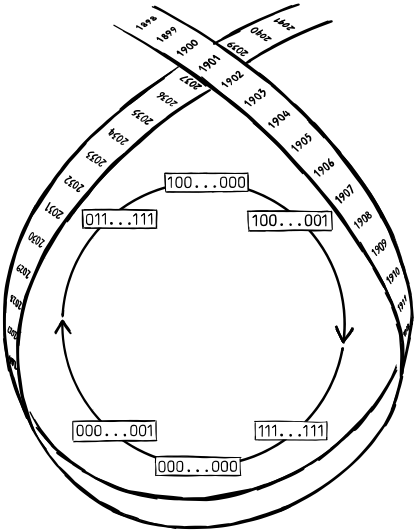
A skilled reverse engineer can still use the machine code to gain knowledge about the software. This can be used to detect security flaws, which can then either be reported to the programmers for them to eliminate or it can be exploited maliciously. Additionally, *Reverse engineering* is used to change old programs to make them compatible with new hardware or to decipher old file formats.



“Brute Force” means you are weak

For some problems in Computer Science, it can be proven that there are no “efficient” ways of solving them. In this case, the only resort is **brute force**. Brute force means that you let the computer try out every possible –and we really mean *every* single, possible and potential– value that solves the problem, until you succeed.

Several cryptographic algorithms are based on such problems where the secret key can only be guessed or found by exhaustively trying out all possible values, which could take a few billion years. Confronted with such time scales, attackers find themselves in a very weak position.


100...000
011...111
100...001
000...001
111...111
000...000

No mercy: the year 2038 problem

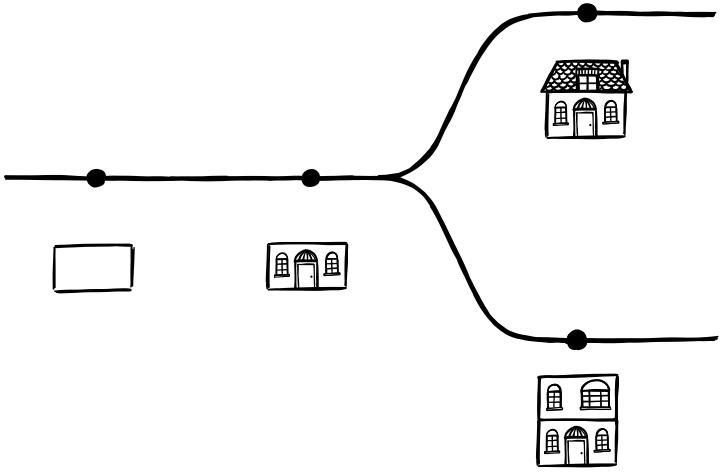
The date of Jan 1, 1970 is to UNIX developpers what year 0 is for Christians, just that in UNIX time you count in seconds, not in days as we do with calendars. UNIX time is commonly used in many modern operating systems.

Now here is a problem:

- in UNIX time, two billion seconds *after* Jan 1, 1970 is Jan 19, 2038
- in UNIX time, two billion seconds *before* Jan 1, 1970 is Dec 13, 1901

Many systems can hold time values only up to four billions, and then values start to "wrap around". This means that on Jan 19, 2038, they will think it is the year 1901!

The clock is now ticking for finding all these machines and to update their software, giving them bigger counters for storing UNIX time. For sure, some systems will be missed and some systems will be too old to be fixed. *Real* time will have no mercy ...



Why do programmers need forks?

Software development is a continuous process. Programmers, engineers, and users work together to improve the current version of a software.

In open-source projects, the source code of the software is openly available. Everyone can look at it and submit their own contributions. Usually, software projects are happy to receive feedback from the community.

Sometimes, the suggested changes deviate too much from the goal of the project. Developers can then decide to copy the source code and start a new independent project. This deviation from the original project is also called a “fork”.

Seminar participants (spring semester 2019)

Fabricio Arend Torres
Nils Bühlmann
Rebecca Dold
Simon Dold
Omnia Kahla
Patrick Kahr
Marcel Lüthi
Sebastian Philipps
Linard Schwendener
Tim Steindl
Christian Tschudin
Marco Vogt
Moira Zuber

List of Topics

0 Hello World
1 Handshake
2 Automation
3 Binary system
4 Run time complexity
5 Finite state machine
6 Turing machine
7 Polygon mesh
8 XOR
9 Monte Carlo method
10 Turing test
11 ASCII
12 System vs service uptime
13 Overflow
14 Camel case
15 Zero vs Null
16 Programming bugs
17 Patching
18 Octal system
19 $25_{dec} = 31_{oct}$
20 Version control systems
21 Interface in software engineering
22 Deadlock
23 Numeric precision
24 Mutex
25 Quine
26 Greedy algorithms
27 Speculative execution
28 Reverse engineering
29 Brute force attack
30 Year 2038 problem
31 Forking open source projects