

Programming

Loriano Storchi

loriano@storchi.org

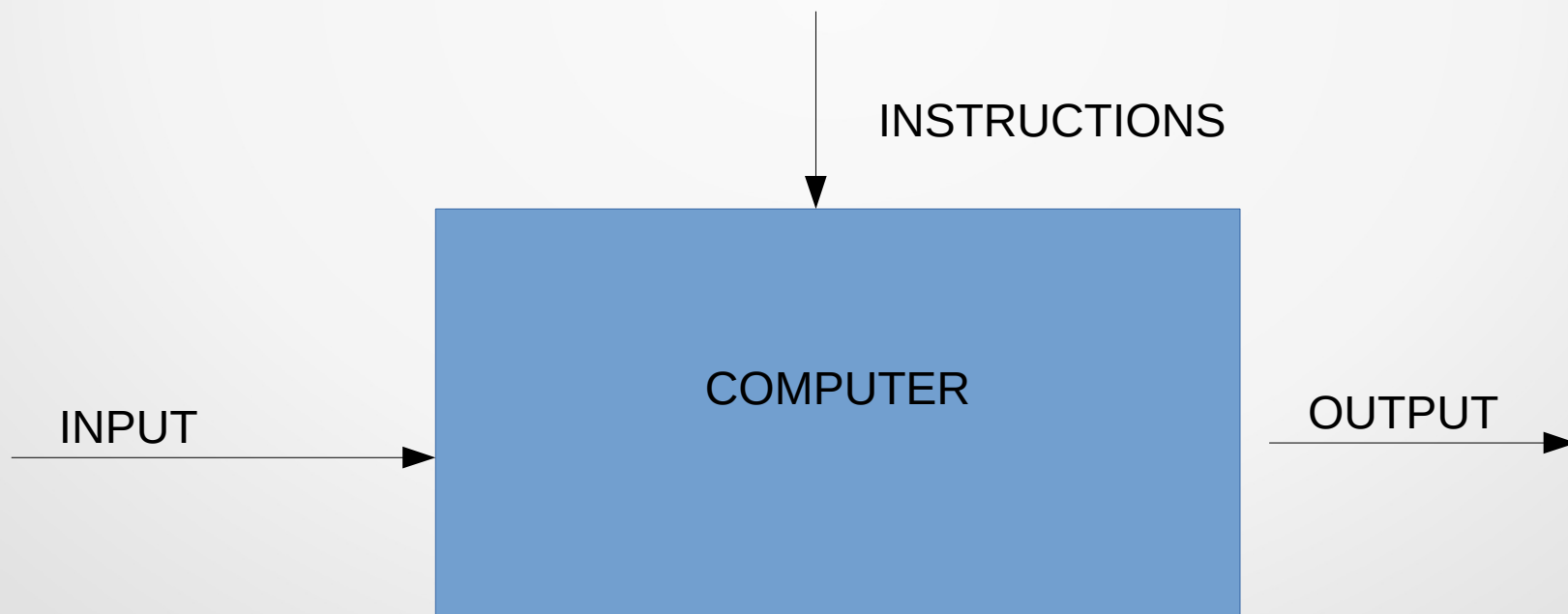
<http://www.storchi.org/>

Algorithms

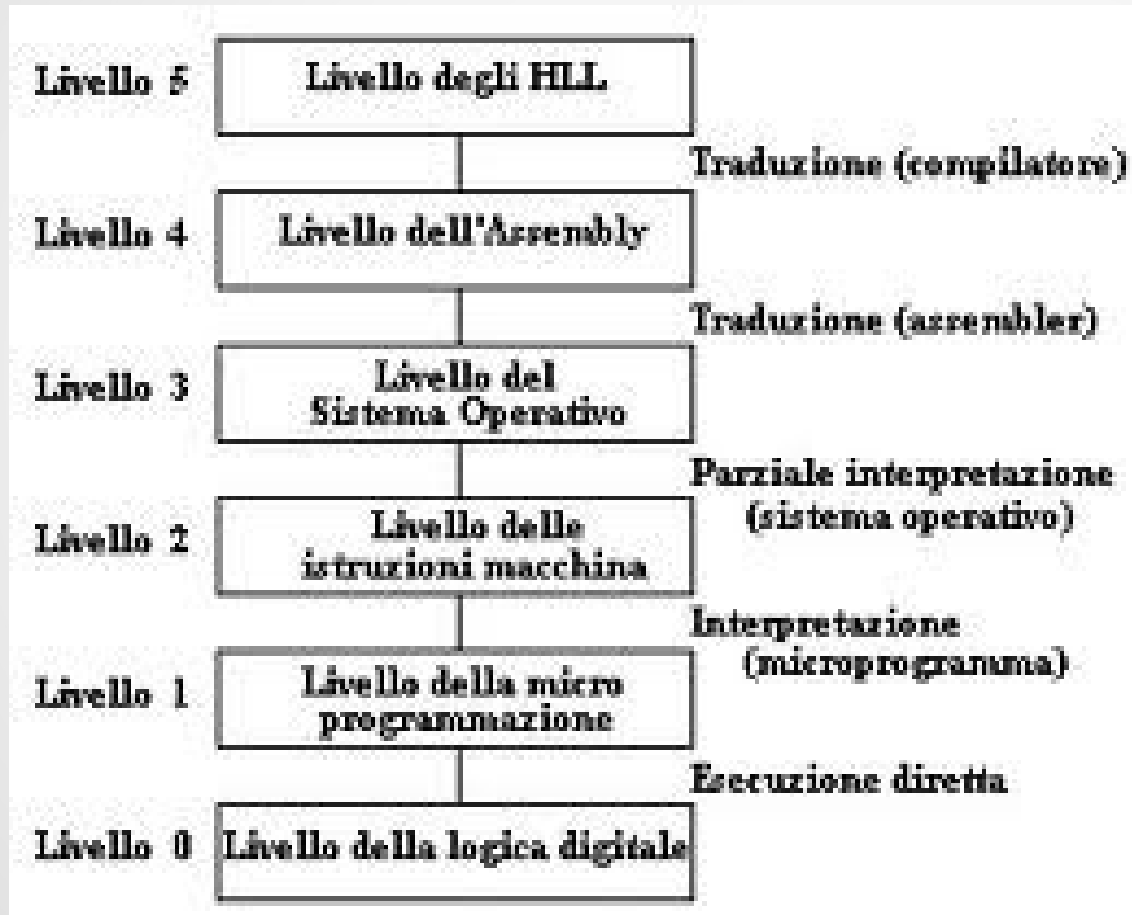
- Algorithms describe the way in which information is transformed. Informatics takes care of their theory, analysis, planning, their efficiency, realization and application.
- An algorithm is a **formal process** that solves a given problem through a **finite number of steps**. The term derives from the Latin transcription of the name of the Persian mathematician al-Khwarizmi, who is considered one of the first authors to refer to this concept. **The algorithm is a fundamental concept of computer science**, first of all because it is the basis of the theoretical notion of **calculability: a problem can be calculated when it can be solved using an algorithm.** (Wikipedia)

Programming

- It identifies the activity by which one **"instructs" a computer to execute a particular set of actions**, which act on input data, in order to solve a problem and thus produce appropriate output data. **Implementation of a given algorithm.**



Languages



The L0 level represents the real computer and the machine language that it is able to execute directly. Each higher level represents an **abstract machine**. The programs (instructions) of each higher level must be or translated in terms of instructions from one of the lower levels, or interpreted by a program that runs on an abstract machine of a strictly lower level.

Machine language and ASSEMBLY

Each computer is able to interpret a low-level language called machine language, the instructions (OPCODE) are simple sequences of bits that the processor interprets and follows a precise series of operations. Each instruction at this level is constituted by extremely basic operation.

To make programming more immediate, the first step was the introduction of the **ASSEMBLY**, where the **binary code is replaced by mnemonic instructions** that are humanly more easily usable.

Machine Language:

```
00000000000000000000000001000000
00000000000100000000000001000100
00000010000000001000000000000000
000000010000000000000000000111100
```

PSEUDO-ASSEMBLY:

```
z : INT;
x : INT 8;
y : INT 38;
LOAD R0,x;
LOAD R1,y;
ADD R0,R1;
STORE R0,z;
```

LANGUAGES

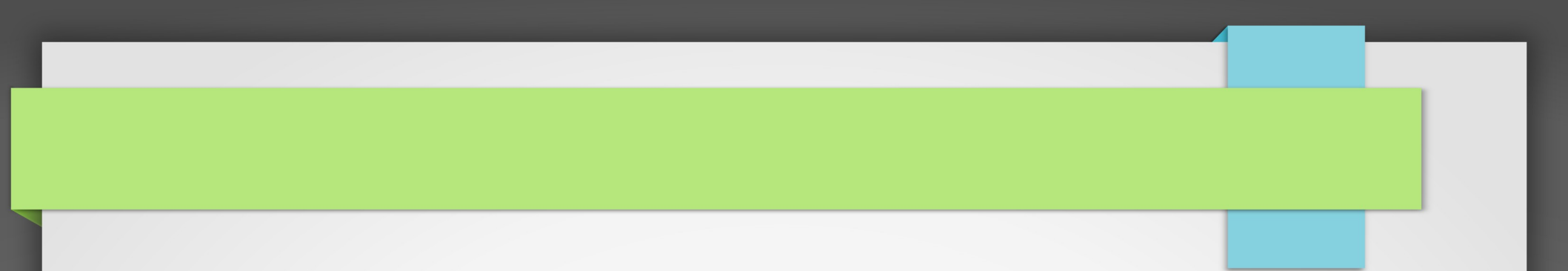
- Today there are many programming languages. In general, every language is more or less adapted to a specific purpose.
- **Natural languages:** they are spontaneously given and are extremely expressive but ambiguous: “**la vecchia porta la sbarra**”
- **Artificial languages:** they are languages that have a precise date of birth and a list of authors. Artificial languages can be formal and non-formal.
- **Formal languages: unambiguous constituted by a finite set of strings built from a finite alphabet.** It is a language for which the form of the sentences (**syntax**) and the meaning of the same (**semantics**) are defined in a non-ambiguous way. It is therefore possible to define an algorithmic procedure able to verify the grammatical correctness of the sentences.

LANGUAGES

- To define a language rigorously some basic tools are needed:
 - **Alphabet**: set of basic symbols necessary to form words
 - **Lexicon**: set of rules necessary to achieve the words of a language (**vocabulary**)
 - **Syntax (grammatical rules)**: set of **rules** that establish **whether a sentence (set of words) is correct**
 - **Semantics**: **defines the meaning of a syntactically correct "sentence"**, for example: `int a [5]`; in C language it allows to reserve space in memory necessary to contain 5 integers

LANGUAGES

- We already mentioned artificial languages, machine languages and low level language (ASSEMBLY)
- **High-level languages** move away from the logic of the processor and are built to be **simple, efficient and readable, as well as independent of the machine**
- There are many programming languages, even if those actually used are about ten.
- The following is a summary classification of these languages
 - C, C ++, C #, JAVA, PYTHON, FORTRAN, PASCAL, BASIC, Objectice-C and many others
- Clearly every programming paradigm is more or less suitable for a more or less specific purpose.



CLASSIFICATION OF LANGUAGES

IMPERATIVE LANGUAGES

- The fundamental component of the program is the instruction, and **each instruction indicates the operation to be performed.** The individual instructions that operated on the program data.
- **The instructions are executed one after the other**
- **Each program consists of two fundamental parts: the declaration of data and the algorithm intended as a sequence of operations**
- **Each instruction is an order** (declarative programming the program is a series of statements)
 - In fact, from a syntactic point of view, many imperative languages use verbs in the imperative (i.e. PRINT, READ, ...)

IMPERATIVE LANGUAGES

```
READ *, A, B
```

```
C = A + B
```

```
PRINT C
```

Then a series of instructions read A and B, calculate C as the sum of A plus B and finally print the result

PROCEDURAL PROGRAMMING

- We can consider it a sub-paradigm of imperative programming.
- **The concept of sub-program (subroutine) or functions is introduced.**
- Then we introduce the **possibility to create portions of source code useful to perform specific functions.**
- These subprograms can **receive input parameters and return output values.**

PROCEDURAL PROGRAMMING

```
SUB EXSUMMA (A, B, C)
```

```
    C = A + B
```

```
END SUB
```

```
FUNCTION SUM (A, B)
```

```
    C = A + B
```

```
    RETURN C
```

```
END FUNCTION
```

```
MAIN
```

```
    READ A, B
```

```
    PRINT SUM (A,B)
```

```
    EXSUM (A,B,C)
```

```
    PRINT C
```

Example of use of a subroutine and a function for calculating the sum (reusability of the code, function libraries)

STRUCTURED PROGRAMMING

- We can consider it a sub-paradigm of imperative programming.
- In practice, **the programmer is bound to use canonical control structures that do not include the unconditional jump instructions (GOTO)**. Thus the syntax of language prevents the use of structures that do not follow certain constraints. (not only)
- The use of the **GOTO instruction inevitably leads to a poor readability of the code** (spaghetti-code)

STRUCTURED PROGRAMMING

- Example

```
10 dim i
20 i = 0
30 i = i + 1
40 if i <= 10 then goto 70
50 print "Programma terminato."
60 end
70 print i & " al quadrato = " & i * i
80 goto 30
```

```
function square(i)
    square = i * i
end function
dim i
for i = 1 to 10
    print i & " al quadrato = " & square(i)
next
print "Programma terminato."
```

Object-Oriented programming

- We can consider it a sub-paradigm of imperative programming.
- This programming paradigm **allows to define Software Objects able to interact with each other.**
- The organization of the software in the form of **objects allows an easier reuse of the same code.** A better organization of large projects.
- The **OOP languages provide for the grouping of part of the source code into classes, each class includes data and methods (functions) that operate on the data themselves.** Classes are abstract models that are invoked at the time of execution to create or instantiate software objects.
- An object-oriented language allows to implement three basic mechanisms using the native language syntax: **encapsulation, polymorphism, inheritance.**

Object-Oriented programming

OBJECTS

CLASSE QUADRATO

ATTRIBUTI

LATO

COLORE

METODI

REAL OTTIENI_AREA()

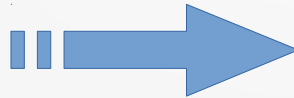
SET_LATO (VAL)



QUADRATO 1

LATO = 1.0

COLORE = VERDE



QUADRATO 2

LATO = 1.5

COLORE = GIALLO

Object-Oriented programming

- **Encapsulation:** precise separation between implementation and interface of the class. **Who uses the class (object) does not have to know the implementation detail.** Use the class using public methods and data interacting with the object without knowing the implementation details

MAIN

```
TRIANGOLO T1
```

```
T1.COLORE = GRIGIO
```

```
T1.SET_LATO(2.0)
```

```
PRINT T1.CALCOLA_AREA ()
```

Object-Oriented programming

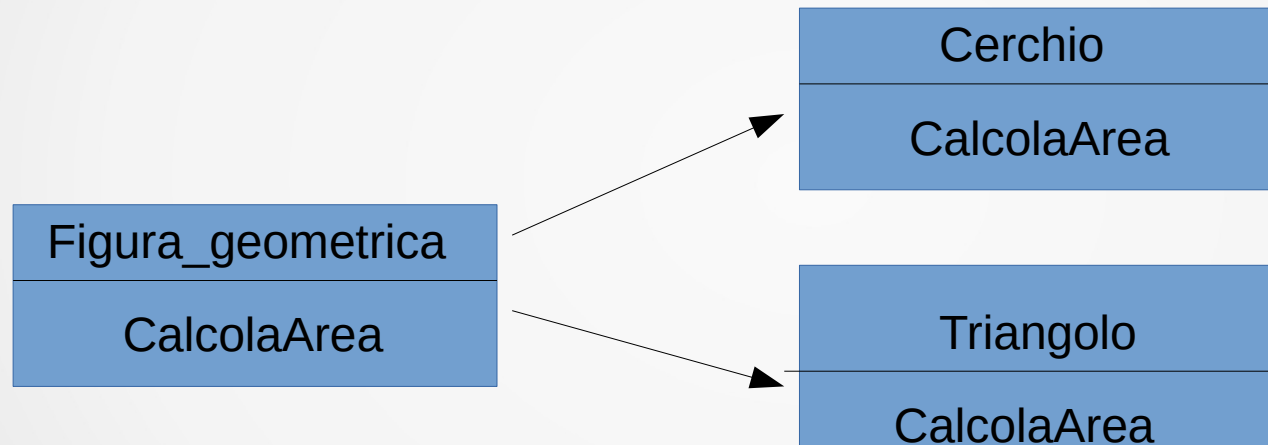
- **Inheritance: a class can inherit from a base class and evolve or specialize its functionality.**
- For example, I can imagine a **figura_geometrica** class from which classes such as **triangolo**, **cerchio**, **quadrato** ... derive.
- Classes that derive from a base class inherit all the methods and properties of the base class, but can specialize by defining their own methods and data.
- **For example, if B is a subclass (or more generally a subtype) than A, any program / function that can use A can also use B**

Object-Oriented programming

- **Polymorphism:** we can try to exemplify this concept by saying: multiple definition of the same function (overloading), classes and parametric functions with respect to the type of data. Simple example of overloading of functions.
- We can formally distinguish in at least four types of polymorphism: by inclusion, parametric, overloading, coercion.
- We will only do a few simple examples useful for clarifying the general concept.

Object-Oriented programming

- we imagine the usual `figura_geometrica` class from which we have derived two classes `cerchio` e `triangolo`



- When the user calls the method `calculate area` this will perform a certain action (calculation of the area in the two cases while having the same name)

FUNCTIONAL PROGRAMMING

- As the name implies, **the execution flow takes the form of evaluation of a series of evaluations of mathematical functions.** The program is therefore a set of functions
- In pure functional languages there is no concept of allocation, or explicit allocation of memory.
- **Values are not found by changing the status of the program, there is no value assignment, but building the new state by functions from the previous state.**
- Uses in the ambit of AI (little or no use in industry)
- Functions can be passed as parameters and returned as "result" from other functions.

DECLARATIVE PROGRAMMING

- Compared to the imperative paradigm, **the program consists of a series of statements and not of orders.**
- In the program **you specify WHAT you want to get not the HOW.** The how is left to the executor
- In practice, **the program (or its execution) can be considered as the demonstration of the truth of an affirmation.**

LANGUAGES

- The languages can also be classified according to the data typing: static typing and dynamic typing.
- **Static typing:** the programmer is forced to explicitly specify the type of each syntactic element. For example, **he must specify the type of a variable and the language will then guarantee that that variable will be used consistently with the declaration.**
- **Dynamic typing:** for example in this case the data will assume a type that varies at runtime depending on the assignments made (see Python)
- We can then also distinguish between **weak and strong typing**

LANGUAGES

- **Parallel programming** languages or paradigms (for modern architectures) if you are curious you can see here for example <http://www.storchi.org/lecturenotes/acr/index.html>
- **Esoteric languages:** these are highly complex and unclear languages. Popular only among the most skilled and used users for the sole purpose of testing the ability of programming (**essentially recreational purpose**)
- **Scripting: originally created for use in Unix shells.** They are languages used to automate repetitive and long tasks.



PROGRAMMING, COMPILING AND LINKING

Programming

- The source is written in ASCII text files. **The source expresses the algorithm implemented in the chosen language.** To write the source you can use simple text editors (VI, Emacs). Or **IDE developmental ambiguities integrated with other tools, such as compilers, linkers and debugger.**
- **Compilation: the source is translated (from the compiler) from a high level language to executable code.** The **advantage** is that the execution is "fast" and that the code is optimized for the specific platform. The **disadvantage** is that you will have to **re-compile for each different operating system or hardware.**
- **Linking:** each program generally uses one or more libraries and the linker links libraries and the starting program together. Linking can be both static and dynamic (for example .so libraries in Linux / Unix-like or .dll in windows)

Programming

```
[redo@buchner csmall (master)]$ cat forloop.c
#include <stdio.h>

int main (int argc, char ** argv)
{
    int i;

    for (i=0; i<10; ++i)
    {
        printf("%d \n", i);
    }

    return 0;
}
[redo@buchner csmall (master)]$ gcc -o forloop forloop.c
[redo@buchner csmall (master)]$ file forloop
forloop: ELF 64-bit LSB shared object, x86-64, version 1 (SYSV), dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2,
for GNU/Linux 3.2.0, BuildID[shal]=88f7e519b5b3fe1e8aed07a28d97a95519dc2354, not stripped
[redo@buchner csmall (master)]$ ldd forloop
        linux-vdso.so.1 (0x00007f3e3f0c5000)
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f3e3e8ad000)
        /lib64/ld-linux-x86-64.so.2 (0x00007f3e3e3eea0000)
[redo@buchner csmall (master)]$ cat forloop
HH/lib64/ld-linux-x86-64.so.2GNUGNU#####T?
![] j "libc.so.6printf__cxa_finalize__libc_start_mainGLIBC_2.2.5_ITM_de
#####H#####e__gmon_start__ITM_registerTMCloneTableui      3
#####
#####
#####
#####
#####
#####H#####TL#####
#####
UH#####
H9#####H#####
]#####f.#####Y
#####R
UH#####H#####
#####]#####f.#####
#####]#####DUH#####H#####
#####AWAVI#####AUATL#####UH#####SA#####)#####
```

Programming

- **Interpretation:** in order to avoid the portability of the programs, the concept of interpretations has been used. In this case the **source code is not compiled "translated" but executed by an interpreter.** This introduces other problems such as performance.
- **Bytecode, P-code:** we can define it as an intermediate approach in which the source program is "translated" into an intermediate code that is interpreted by a virtual machine. This allows to combine two advantages a good speed of execution together with an extreme portability of the program. **JIT (Just in Time) at the time of execution fill in the intermediate code in machine code.**

Programming

```
[redo@buchner helloworld (master)]$ cat hello.py
print "Hello World"
[redo@buchner helloworld (master)]$ python hello.py
Hello World
[redo@buchner helloworld (master)]$ ls
hello.py  oltrehw1.py  oltrehw2.py  oltrehw3.py  oltrehw4.py
[redo@buchner helloworld (master)]$
```

```
└─ $ python3 -m compileall hello.py
Compiling 'hello.py'...
[redo@buchner /home/redo/Lezioni/Programmazione_Informatica/teaching_github/helloworld (master)
└─ $ cat ./__pycache__/hello.cpython-36.pyc
3
Hello WorldN)print hello.py<module>
[redo@buchner /home/redo/Lezioni/Programma
hub/helloworld (master)
└─ $ od ./__pycache__/hello.cpython-36.pyc
0000000 006463 005015 021320 057234 000026 000000 000343 000000
0000020 000000 000000 000000 000000 001000 000000 040000 000000
0000040 071400 000014 000000 000145 000144 000603 000001 000544
0000060 000123 001051 005572 062510 066154 020157 067527 066162
0000100 047144 000451 002732 071160 067151 124564 071000 000002
0000120 000000 001162 000000 175000 064010 066145 067554 070056
0000140 155171 036010 067555 072544 062554 000476 000000 071400
0000160 000000 000000
0000164
```



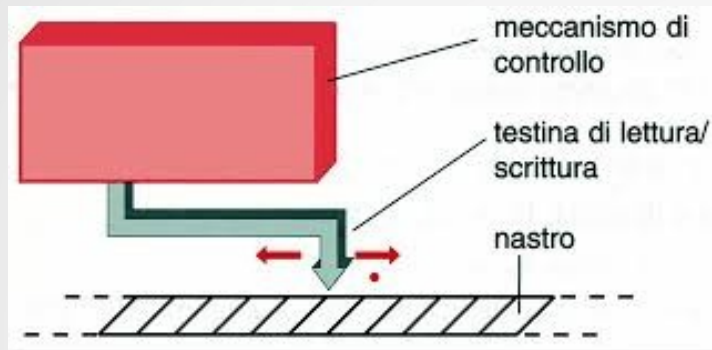
COMPUTABILITY AND COMPLEXITY

COMPUTABILITY AND COMPLEXITY

- **Computability** : Given a function it is called calculable if we can find an algorithm (hence a procedure that mechanically executes a finite number of steps) that computes it.
- **Church-Turing Thesis**, the class of calculable functions coincides with the class of functions that can be calculated by a Turing machine.
- All calculation machines (computers) can be traced back to a Turing machine.

COMPUTABILITY AND COMPLEXITY

- **Turing Machine**



Deterministic model with ribbon and 5-field instructions:

- 1 - A long ribbon at will that can contain characters or empty spaces
- 2 - head / device for reading and writing with which to read and write on the tape and obviously the head can move the tape to the right or left
- 3 - The machine has an internal status

At every step, the machine reads a symbol and, depending on its internal state, can change state and then write a symbol on the tape and then move the tape to the right or left.

The behavior of the Turing Machine is programmed defining the rules or quadruples of the type:

(internal state, symbol-read, new-state, written-symbol / direction)

COMPUTABILITY AND COMPLEXITY

- **Halting problem:** given a certain program and given an input it is impossible to determine if this program will end or not. **(this problem is strongly linked to Gödel's incompleteness theorem)**

COMPUTABILITY AND COMPLEXITY

- **Algorithm complexity:** it is the measure of the difficulty of a calculation (algorithm + input)
- **The goodness of an algorithm is evaluated in relation to the time and space necessary for its execution**, in general therefore according to the resources required.
- **Clearly the execution time is a function of the type of input as well as the type of hardware used, so it makes no sense to classify the algorithms according to the number of seconds required for its execution.**
- **The calculation time is therefore expressed as the number of elementary operations according to the N dimension of the input data**

COMPUTABILITY AND COMPLEXITY

- Example calculation of the efficiency of an algorithm in which we **look for the minimum m within a set of N numbers $\{x_1, x_2, \dots, x_N\}$**
- Let's imagine tackling the problem as follows:
 - I choose x_1 as a minimum possible
 - compare it with x_2 , then x_3 and so on
 - If I find a smaller x_i I continue comparing it with that as I did with x_1
 - At the end I found the minimum
- To do everything **I will have done N comparisons so the efficiency of the algorithm is directly proportional to the N dimensions of the input**

COMPUTABILITY AND COMPLEXITY

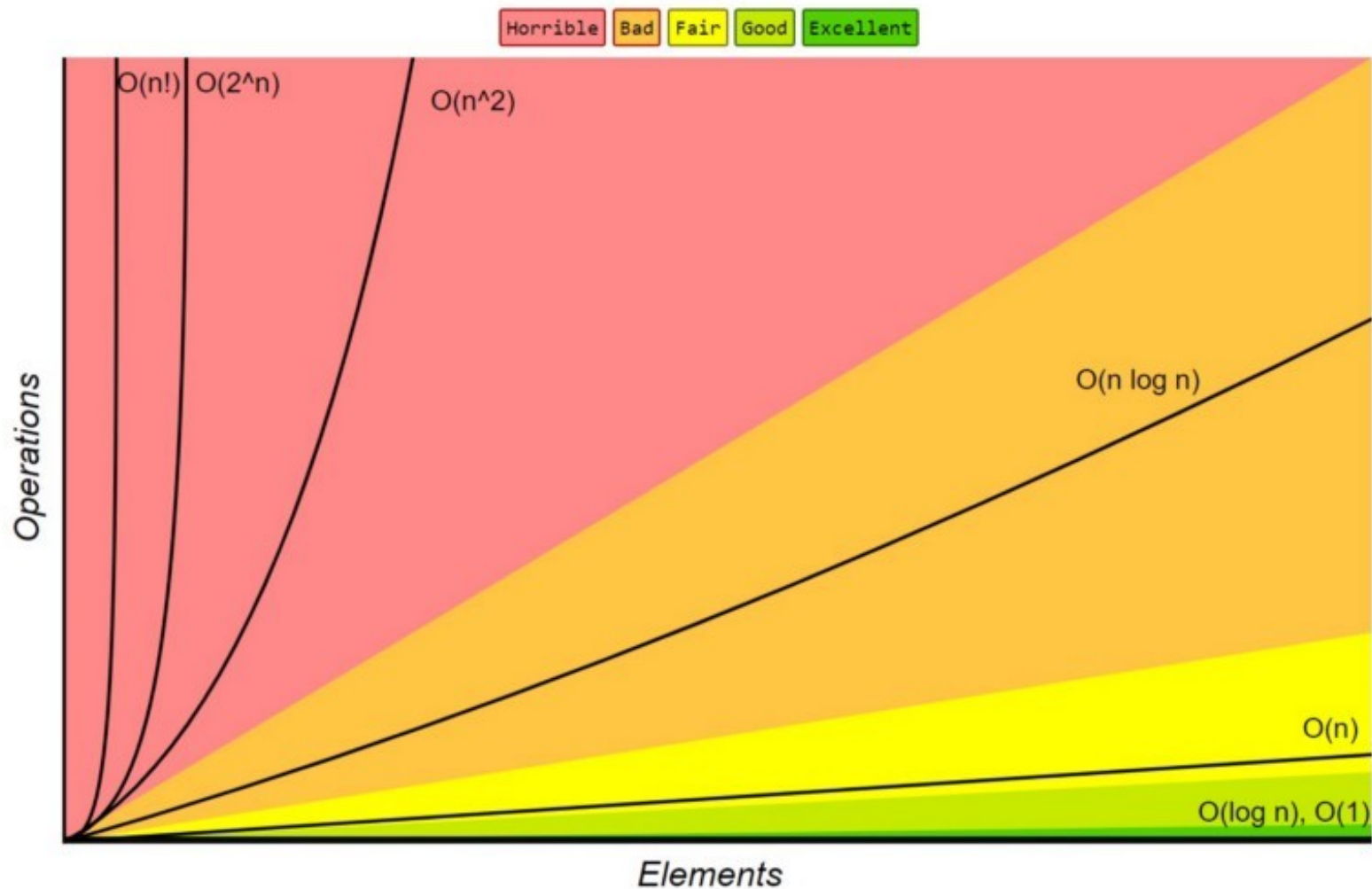
- **The efficiency of a given algorithm can therefore be expressed as a function $f(N)$, hence a function of the variable N which represents the size of the input data.**
 - **This function therefore expresses the number of elementary operations necessary to solve the problem** by means of the algorithm given as a function of the input size
 - It therefore **represents the complexity of the computation**
 - Given N an algorithm A is more efficient than another B if at the growth of N $f_A(N)$ is less than or equal to $f_B(N)$
- **The execution time of a program therefore depends on the complexity of the algorithm, on the size of N and obviously on the "speed" of the machine on which it is executed.**

COMPUTABILITY AND COMPLEXITY

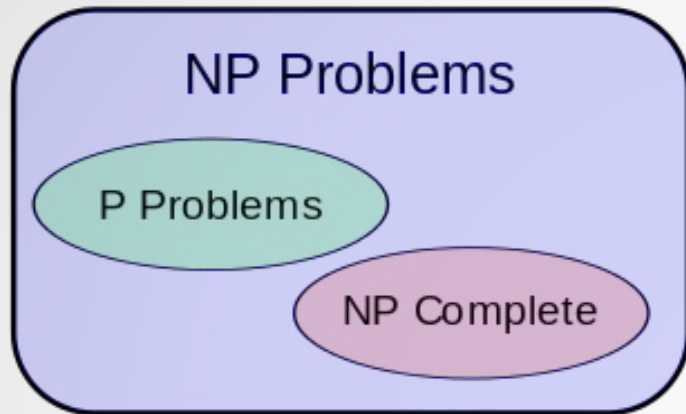
- To subdivide the algorithms into complexity classes, the following criterion is used:
- A function $f(N)$ is said to be of order $g(N)$ and is indicated with $f(N) = O(g(N))$ if there exists a constant K such that, unless for a finite number of values of N the following inequality is always true: $f(N) \leq K * g(N)$. You can also write $f(n) / g(N) \leq K$
- For example $2 * N + 5 = O(N)$ in fact $2 * N + 5 \leq 7 * N$ for every N greater than zero

COMPUTABILITY AND COMPLEXITY

Big-O Complexity Chart



COMPUTABILITY AND COMPLEXITY



Complexity classes P and NP, if P is the same as NP or less, is still an open problem (million dollar problem Clay Math Institute)

Class P problems solvable in a deterministic Turing machine in polynomial times.

Class NP problems for which an algorithm with polynomial complexity is not known. But they are verifiable instead quickly

NP-complete problems The simplest way to describe it if one of the NP-complete problems is easy then all are since I can "convert" the resolution of one into another. Likewise, if one is difficult, they are all difficult.

EXAMPLE: Factorization in prime numbers of an integer NP problem (most likely it is not NP-complete yet we can not say) given c find the factors prime a and b such that $a * b = n$