Fundamentals of programming

Part 1 Introduction to Programming

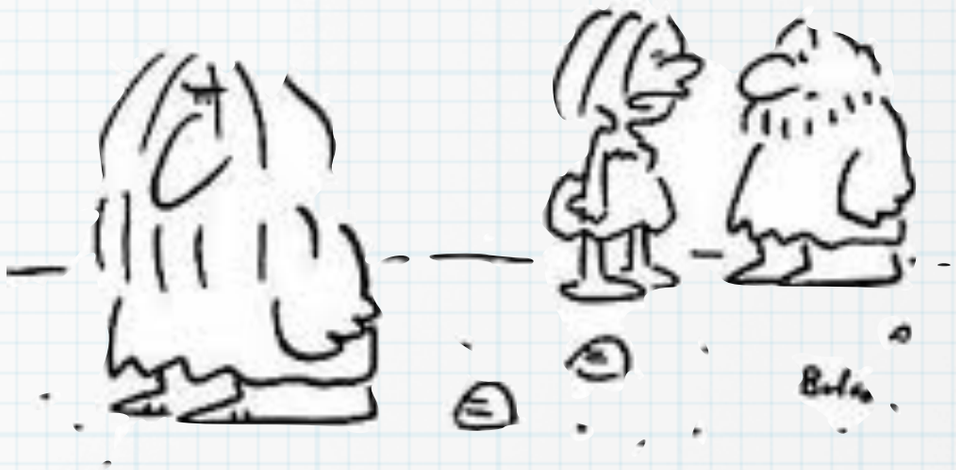
ETSI Informática, Málaga 🇪🇸

Lecturer Dr. Juan Falgueras
Office: 3.2.32

Part I. Introduction to Programming. Contents

I Informatics and the Computer Programming's role

- Informatics as a pure Science
  - It is Linguistics
  - It is Discrete Mathematics



  - It is based on Theory of Computability
- and as a Technology
  - It is the base of Appliances
  - …of Robotics, Mechatronics, etc

Ruling the programming process

- Is programming an art?
- What was the 1968 "*Software crisis*"?
- Today the main problem is dealing with the complexity of large programs (its correction, maintenance, easiness of modification, etc.)

Sizing programs

Programs size is usually measured in number of lines, files, structures, etc.

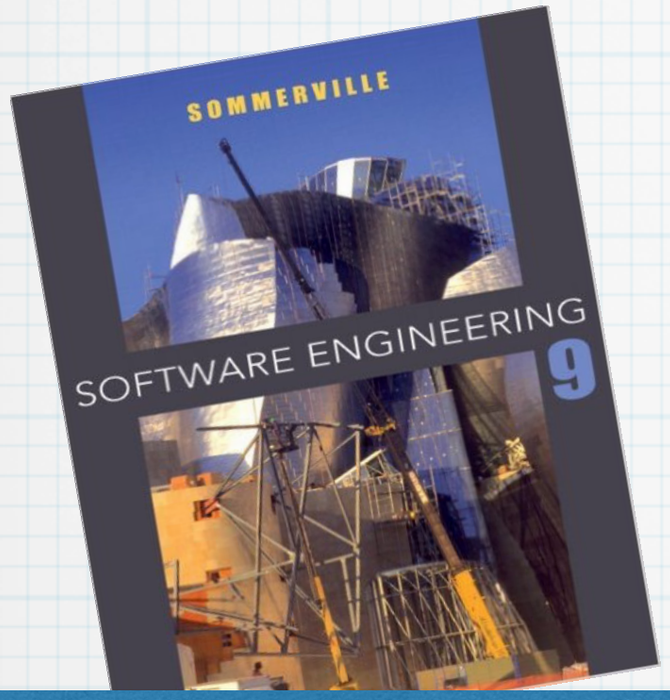
There are

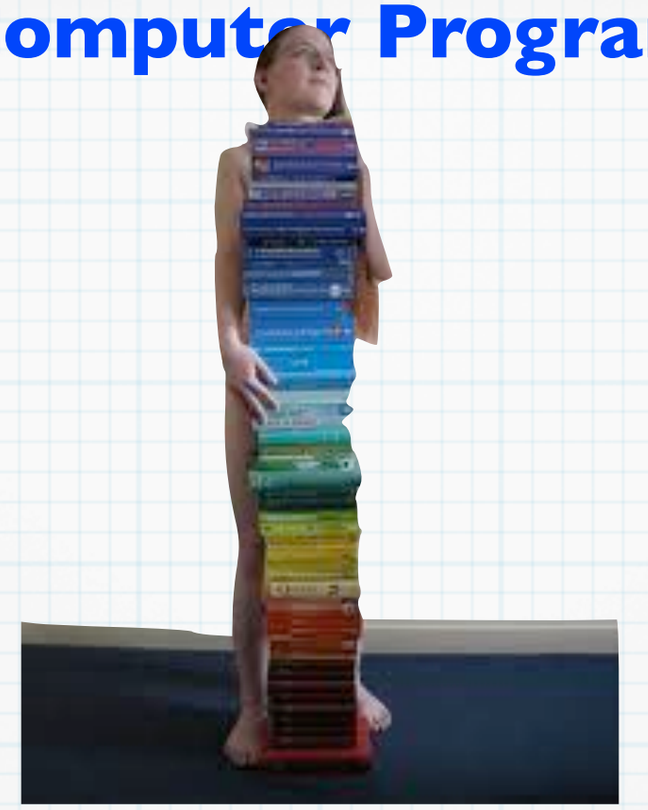
- **small**: few lines
- **medium**: few files or pages and
- **big** programs

It is with the latter for which Software Engineering techniques are important

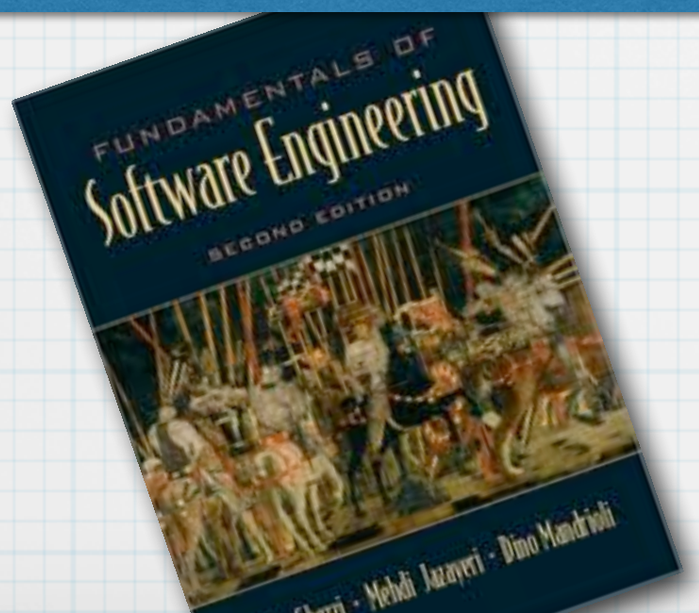
Big programs require: organised specialised teams and strict stages

I Informatics and the Computer Programming's role



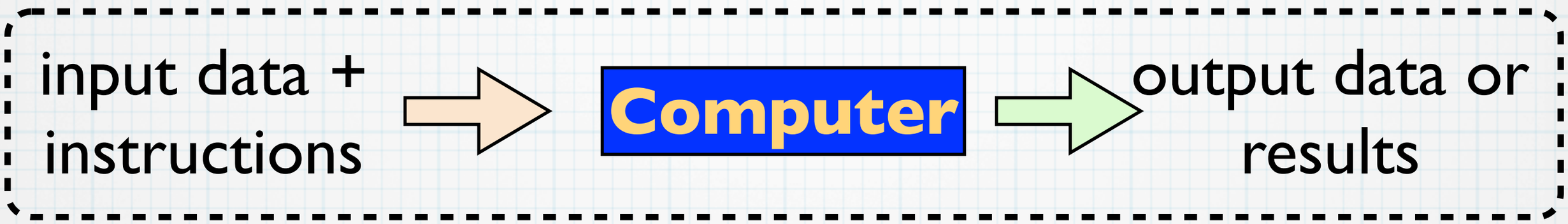


Programming is an activity really present in all areas.
Software Engineering studies techniques to word on programming computers



Part I. Introduction to Programming. Contents

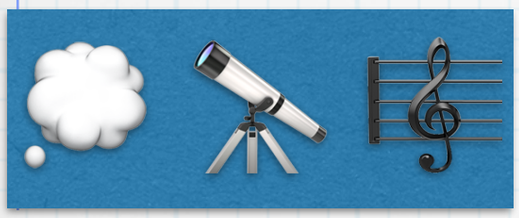
2 The Computer as an Information Processor device



- Is a machine that receives ***inputs*** and produces and ***outputs***, the computer is made of
  - **Hardware**: the physical parts, and
  - **Software**: the electric values that represent *data* and *instructions*

Part I. Introduction to Programming. Contents

Programming language levels



abstract, high or human level

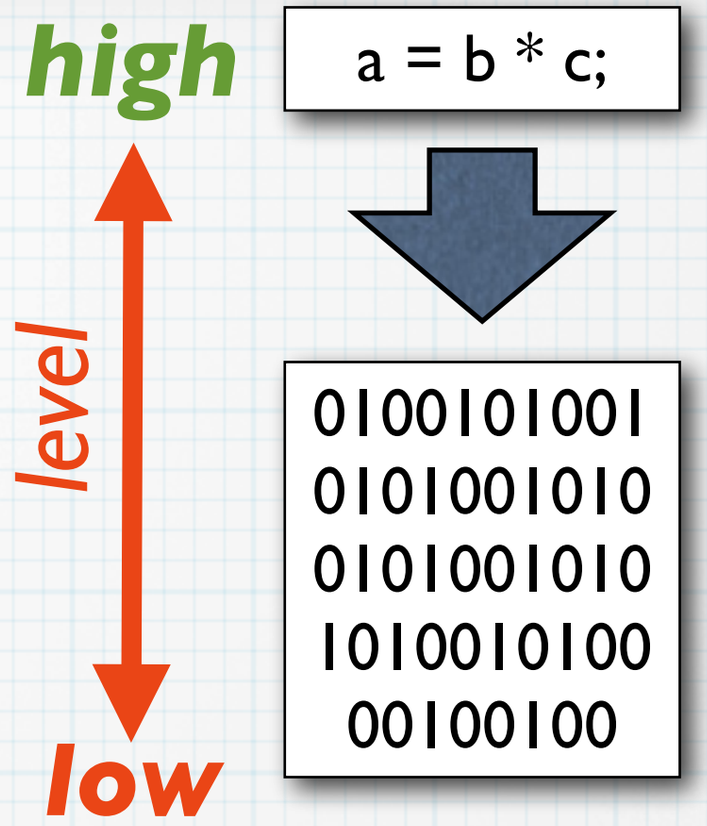
```
a = b * c;
```

```
0100101001
0101001010
0101001010
1010010100
00100100
```

concrete detail, low/machine level

- *Programming Languages* are for humans
- *Processor instructions*, or *machine instructions* or *low-level* instructions, are **un-readable for humans**. They are composed only of 0's and 1's

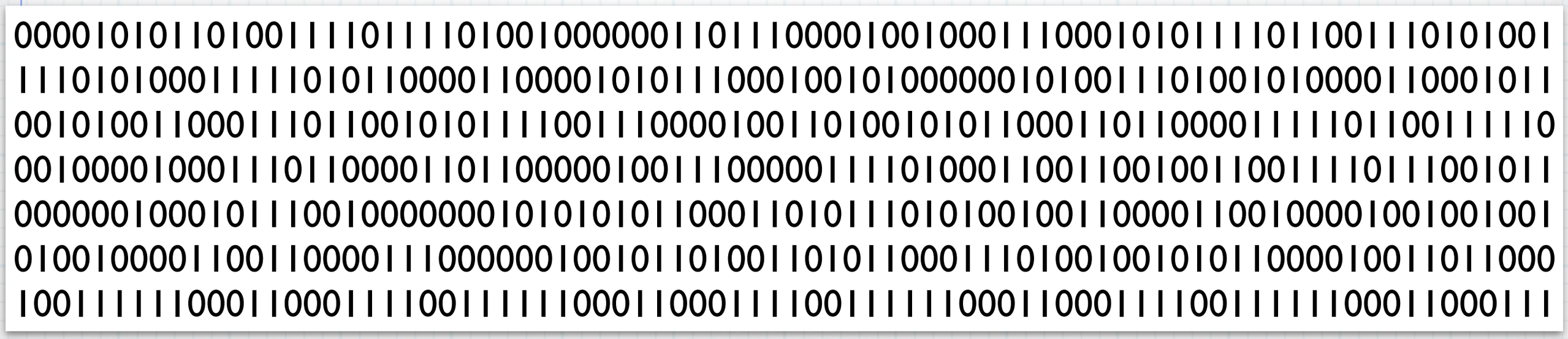
Programming language levels



There are many programming languages…

- ***Natural*** languages (*English* or *Spanish*, the highest levels)
- **high-level** languages: *Mathematica*, *Matlab*, *Pascal*, *C++*, *Java*, *Python*, *Ruby*, etc.
- **low-level** instructions: Assembler

3 Information encoding



- To represent information inside the computer it must be encoded
- Everything is finally represented by 0's and 1's
- **Groups** of these values 0's and 1's can be set up standing for some value

3 How information is encoding? Groups of bits

- **bit**: **bi**nary digi**t**; a bit is ONE binary digit (0 or 1)
  - for example, *true* and *false* can be encoded in 1 bit
- **nybble**: a group of 4 bits (ex: 1010.) It has room for 16 distinct values)
  - A nybble has room for a decimal number, 0…9
- **byte**: 8 bits. For example, the letters of an occidental alphabet fit in a byte. It has 256 possible distinct values (0000 0000…1111 1111)

3 Information encoding. Greater groups of bytes

- Greater groups of **bytes** are called:
  (numbers in bytes. 10^n = 10^n)

1 kB ≠ 1 kb

Decimal		**Binary**		
Name (Symbol)	**Value**	**Value**	**=**	**Name (symbol)**
kilobyte (**KB**)	10^3	2^10	1,024	kilobyte (**Kb**)
megabyte (**MB**)	10^6	2^20	1,048,576	megabyte (**Mb**)
gigabyte (**GB**)	10^9	2^30	1,073,741,824	gigabyte (**Gb**)
terabyte (**TB**)	10^12	2^40	1,099,511,627,776	terabyte (**Tb**)
petabyte (**PB**)	10^15	2^50	1,125,899,906,842,624	petabyte (**Pb**)
exabyte (**EB**)	10^18	2^60	1,152,921,504,606,846,976	exabyte (**Eb**)
zettabyte (**ZB**)	10^21	2^70 10^21	1,180,591,620,717,411,303,424 1,000,000,000,000,000,000,000	zettabyte (**Zb**)
yottabyte (**YB**)	10^24	2^80 10^24	1,208,925,819,614,629,174,706,176 1,000,000,000,000,000,000,000,000	yottabyte (**Yb**)

3 Information encoding.
Number encoding

3.141592 2.728182

- Efficient **number** encoding is especially important

- The *encoding* must be **compact** and allow the set of operations numbers are usually involved in: **+-*/**

- Simple numbers, or natural numbers, are directly encoded as binary$_{(2}$

- Exercise: convert the real number 26.1875 to binary

  $26.1875_{(10} = 11010.0011_{(2}$

3 Information encoding.
Number encoding

- Verify that:

$$26_{(10} = 11010.0011_{(2}$$

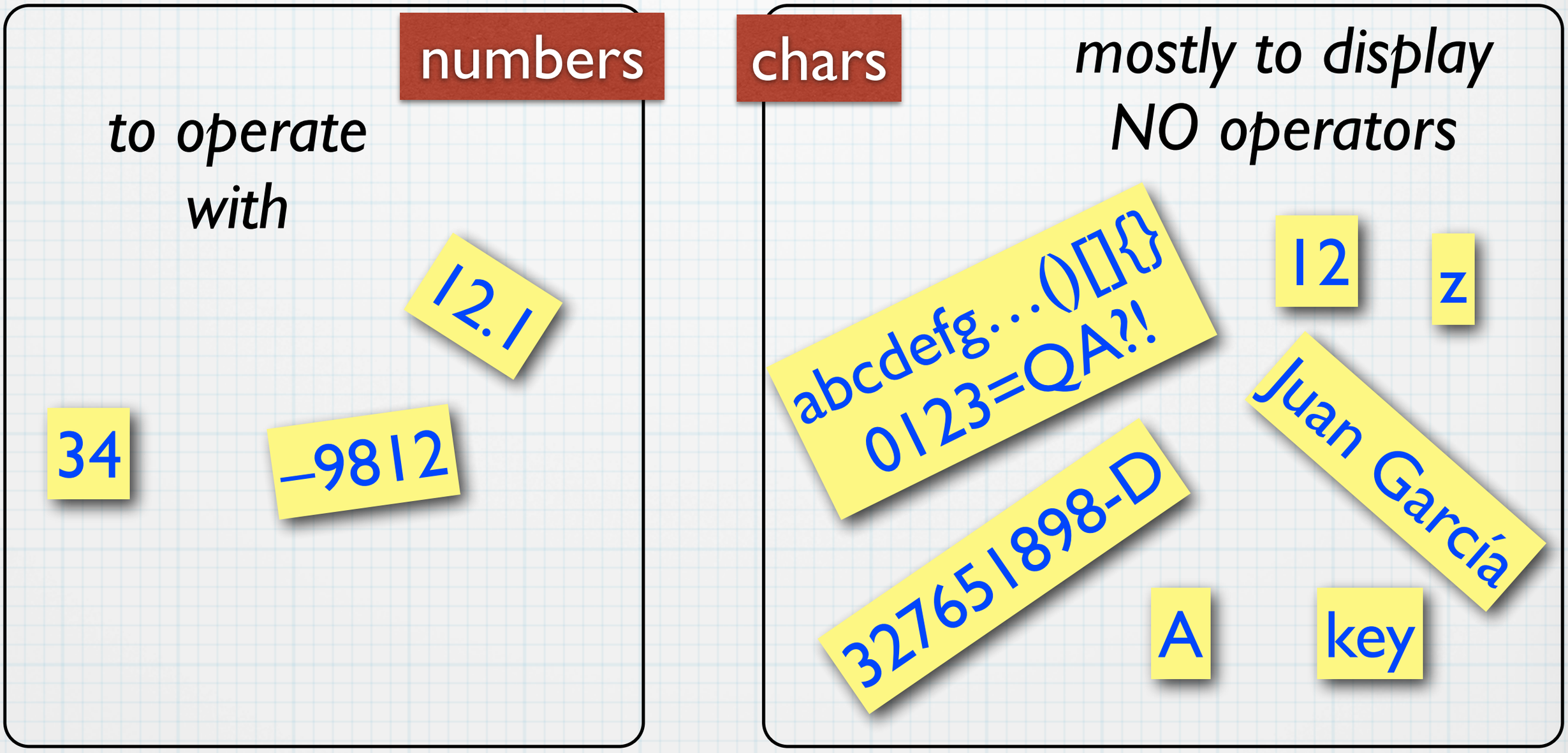
13	1
6	0
3	1
1	1
0	0

0	0.1875
0	0.3750
1	0.7500
1	0.5000

0.1875	0.3750	0.7500	0.5000
x 2	x 2	x 2	x 2
0.3750	0.7500	0.5000	1.0000

3 Information encoding.
Character encoding

- Chars or letters encoding is simpler as you only need to **map** (to correspond) each char with an artificially agreed code number. They haven't got possible **operations**, only are encoded



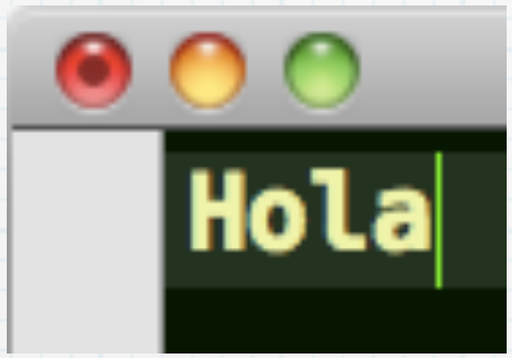
3 Information encoding.
Character encoding

abcdefg…()[]{}0123=QA?!

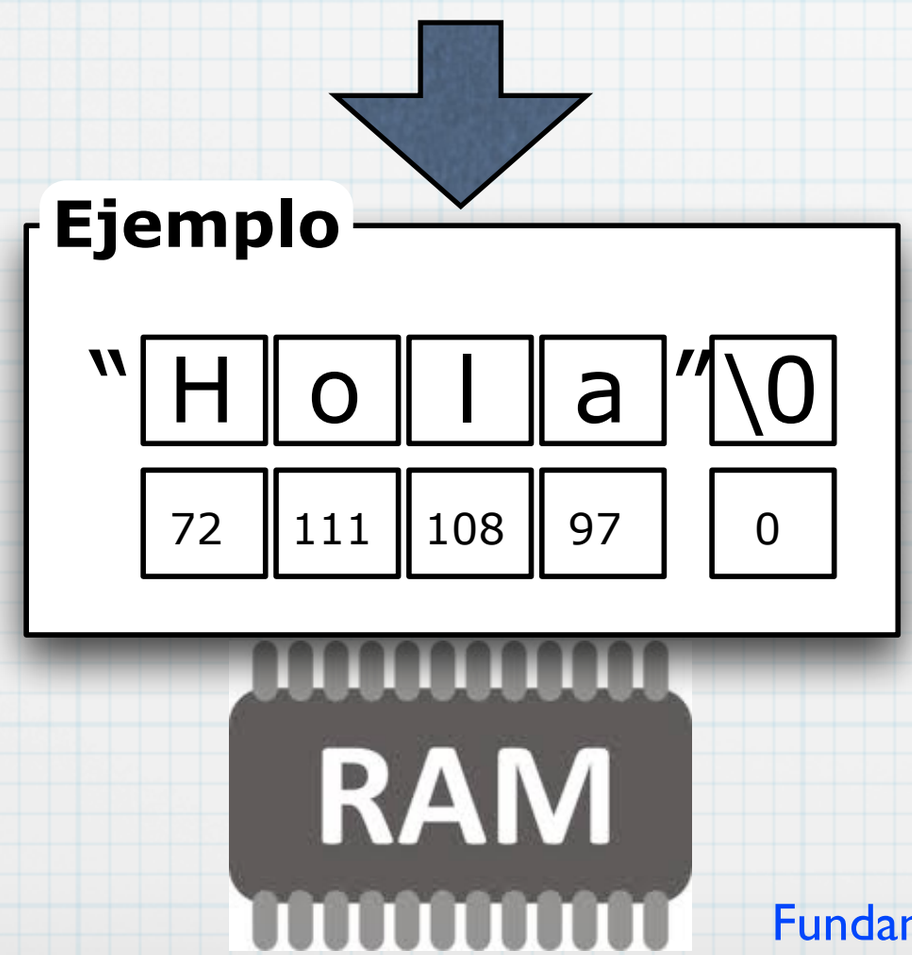
- We need a set of codes that maps each alphabetic letter with a binary value.
- A byte (=256 values) is big enough to gather
  - …26 chars plus capital chars, and
  - many punctuation symbols, as well as
  - *digits* 0 to 9, and still some other special "chars"
- The universal table that represents the number associated which each char is the **ASCII** table
- Do you know what **UTF-8** means?

abcdefg…()[]{}0123=QA?!

- For example, if we write "Hola", it is internally encoded as:



each codes is the corresponding ASCII code of each char

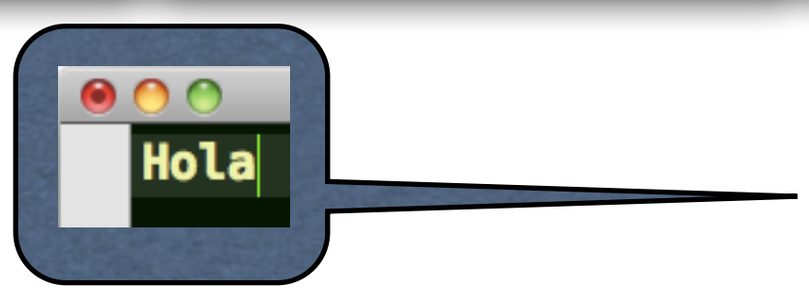


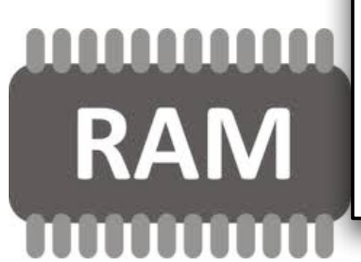
Dec	Octal	Hex	Binary	Value
000	000	000	00000000	NUL (Null char.)
001	001	001	00000001	SOH (Start of Header)
002	002	002	00000010	STX (Start of Text)
003	003	003	00000011	ETX (End of Text)
004	004	004	00000100	EOT (End of Transmission)
005	005	005	00000101	ENQ (Enquiry)
006	006	006	00000110	ACK (Acknowledgment)
007	007	007	00000111	BEL (Bell)
008	010	008	00001000	BS (Backspace)
009	011	009	00001001	HT (Horizontal Tab)
010	012	00A	00001010	LF (Line Feed)
011	013	00B	00001011	VT (Vertical Tab)
012	014	00C	00001100	FF (Form Feed)
013	015	00D	00001101	CR (Carriage Return)
014	016	00E	00001110	SO (Shift Out)
015	017	00F	00001111	SI (Shift In)
016	020	010	00010000	DLE (Data Link Escape)
017	021	011	00010001	DC1 (XON) (Device Control 1)
018	022	012	00010010	DC2 (Device Control 2)
019	023	013	00010011	DC3 (XOFF)(Device Control 3)
020	024	014	00010100	DC4 (Device Control 4)
021	025	015	00010101	NAK (Negative Acknowledgement)
022	026	016	00010110	SYN (Synchronous Idle)
023	027	017	00010111	ETB (End of Trans. Block)
024	030	018	00011000	CAN (Cancel)
025	031	019	00011001	EM (End of Medium)
026	032	01A	00011010	SUB (Substitute)
027	033	01B	00011011	ESC (Escape)
028	034	01C	00011100	FS (File Separator)
029	035	01D	00011101	GS (Group Separator)
030	036	01E	00011110	RS (Request to Send)(Record Separator)
031	037	01F	00011111	US (Unit Separator)

Dec	Octal	Hex	Binary	Value
032	040	020	00100000	SP (Space)
033	041	021	00100001	!
034	042	022	00100010	"
035	043	023	00100011	#
036	044	024	00100100	$
037	045	025	00100101	%
038	046	026	00100110	&
039	047	027	00100111	'
040	050	028	00101000	(
041	051	029	00101001	)
042	052	02A	00101010	*
043	053	02B	00101011	+
044	054	02C	00101100	,
045	055	02D	00101101	-
046	056	02E	00101110	.
047	057	02F	00101111	/
048	060	030	00110000	0
049	061	031	00110001	1
050	062	032	00110010	2
051	063	033	00110011	3
052	064	034	00110100	4
053	065	035	00110101	5
054	066	036	00110110	6
055	067	037	00110111	7
056	070	038	00111000	8
057	071	039	00111001	9
058	072	03A	00111010	:
059	073	03B	00111011	;
060	074	03C	00111100	<
061	075	03D	00111101	=
062	076	03E	00111110	>
063	077	03F	00111111	?
064	100	040	01000000	@

Dec	Octal	Hex	Binary	Value
065	101	041	01000001	A
066	102	042	01000010	B
067	103	043	01000011	C
068	104	044	01000100	D
069	105	045	01000101	E
070	106	046	01000110	F
071	107	047	01000111	G
072	110	048	01001000	H
073	111	049	01001001	I
074	112	04A	01001010	J
075	113	04B	01001011	K
076	114	04C	01001100	L
077	115	04D	01001101	M
078	116	04E	01001110	N
079	117	04F	01001111	O
080	120	050	01010000	P
081	121	051	01010001	Q
082	122	052	01010010	R
083	123	053	01010011	S
084	124	054	01010100	T
085	125	055	01010101	U
086	126	056	01010110	V
087	127	057	01010111	W
088	130	058	01011000	X
089	131	059	01011001	Y
090	132	05A	01011010	Z
091	133	05B	01011011	[
092	134	05C	01011100	\
093	135	05D	01011101	]
094	136	05E	01011110	^
095	137	05F	01011111	_
096	140	060	01100000	`

Dec	Octal	Hex	Binary	Value
097	141	061	01100001	a
098	142	062	01100010	b
099	143	063	01100011	c
100	144	064	01100100	d
101	145	065	01100101	e
102	146	066	01100110	f
103	147	067	01100111	g
104	150	068	01101000	h
105	151	069	01101001	i
106	152	06A	01101010	j
107	153	06B	01101011	k
108	154	06C	01101100	l
109	155	06D	01101101	m
110	156	06E	01101110	n
111	157	06F	01101111	o
112	160	070	01110000	p
113	161	071	01110001	q
114	162	072	01110010	r
115	163	073	01110011	s
116	164	074	01110100	t
117	165	075	01110101	u
118	166	076	01110110	v
119	167	077	01110111	w
120	170	078	01111000	x
121	171	079	01111001	y
122	172	07A	01111010	z
123	173	07B	01111011	{
124	174	07C	01111100	|
125	175	07D	01111101	}
126	176	07E	01111110	~
127	177	07F	01111111	



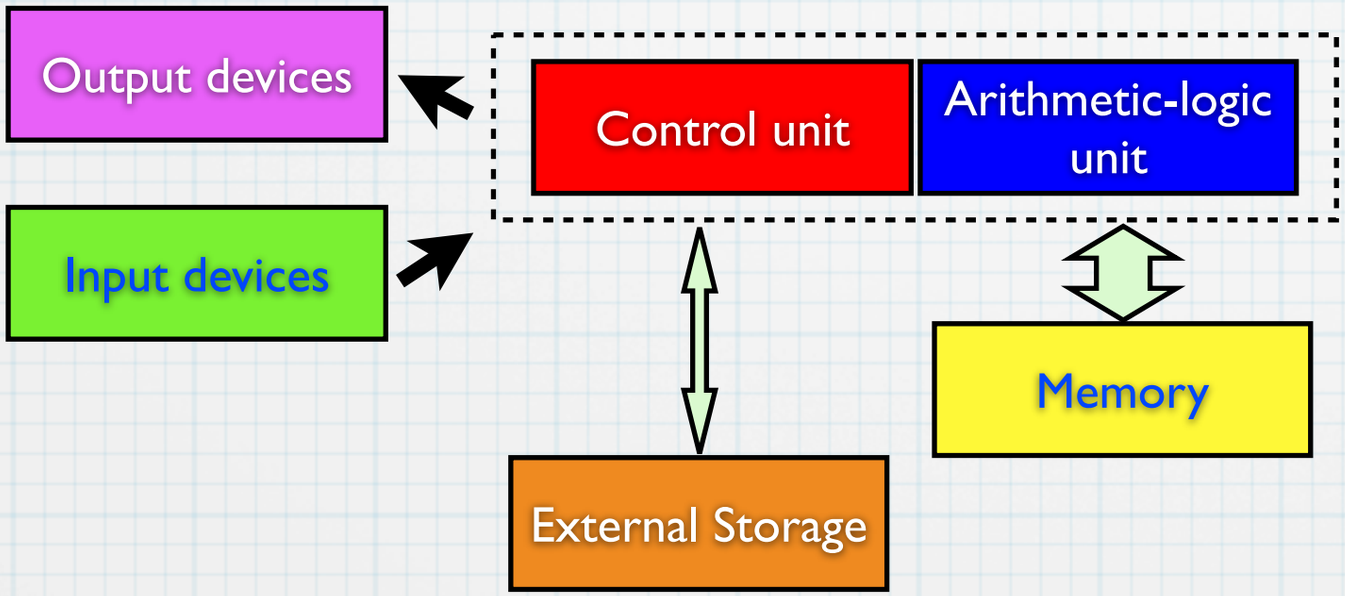


Ejemplo

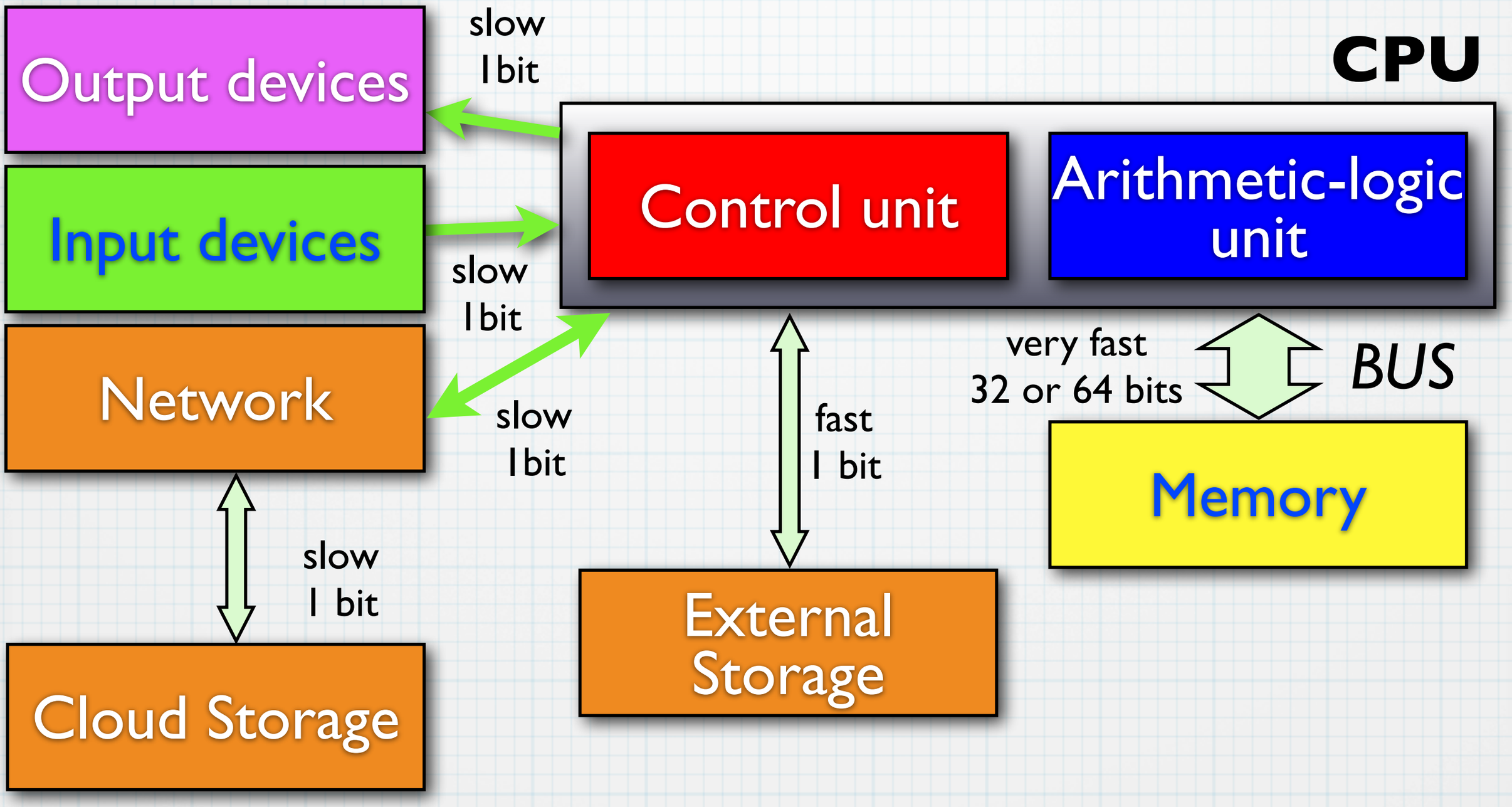
“	H	o	l	a	”	\0
	72	111	108	97		0

Part I. Introduction to Programming. Contents

von Neumann (1940)
architecture has greatly evolved,
but it is still valid



4 Computer's functional structure. Internal functioning

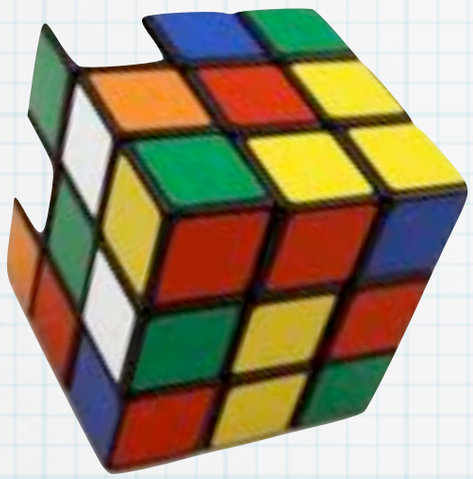


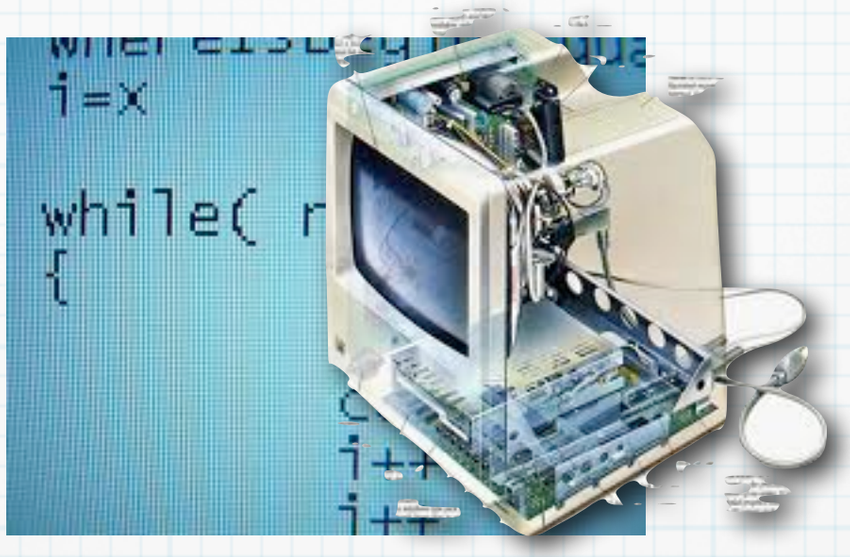
How does it all start?

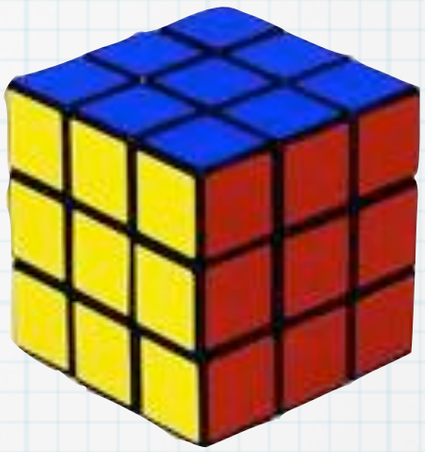
- Application programs are stored in **files**.
- The ***loader*** reads-and-deploys them in RAM changing also objects relative addresses, and making them absolute, among other things…
- Once in memory, a special CPU register called **PC** gets the address of the very start of the program. Program execution starts……
- Each time the program needs to change the instruction to execute, is made **PC** jump accordingly

Part I. Introduction to Programming. Contents

5 Algorithms and Problem Solving







The goal of programming is to write the

correct sequence of instructions

to be executed by a **specific machine**

to solve a **specific problem**

5 Algorithms and Problem Solving. Steps

- In 1945 **George Polya** published the book *How To Solve It* which indeed became his most prized publication and is a classic reference to problem solving

  - Understand the Problem
  - Devising a Plan
  - Carrying out the Plan
  - Looking Back

5.1 What we want. Algorithm concept

al-Khwārizmī
c. 780-850

- An **algorithm** is a **procedure** that has an end

- The **algorithm** concept comes from the idea of *recipe*, a series of steps to solve something.
    - The name comes from a persian mathematician who collected in a book procedures to solve many different problems

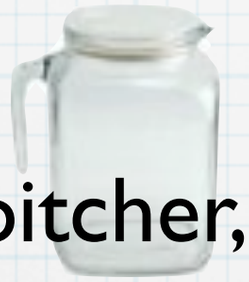
5.1 What we want. Algorithm concept

Example of ***recipe***: Lemonade

Ingredients

- 4 lemons, juiced
- 1 litre water
- 1/2 cup white sugar

Directions



- In a 2 litres pitcher, combine the lemon juice, water and sugar
- Stir until sugar is dissolved
- Chill in refrigerator.

5.1 What we want. Algorithm concept

- **Problem:**
  - To sum the natural numbers from 1 to 100
- **Algorithm:**

```
s ← 0
i ← 1
while i ≤ 100 do
   s ← s + i
   i ← i + 1
out ← sum
```

5.1 What we want. Algorithm concept

```
// suma100.cpp
// juanfc 2017-10-05
//

#include <iostream>
using namespace std;

int main()
{
    int s = 0;
    int i = 1;
    while ( i <= 100 ) {
        s = s + i;
        i = i + 1;
    }
    cout << "Sum 1 to 100 = " << s << endl;
    return 0;
}
```

```
sum ← 0
i ← 1
while i ≤ 100 do
    sum ← sum + i
    i ← i + 1
out ← sum
```

5.2 What can be done. Computability and Complexity

Not every problem, yet well formulated, can be algorithmically solved

- **Computability** deals with this issue, and

- **Complexity** deals with computational efforts algorithms demand from the computers

5.2 What can be done. Computability

- Until the 1930s optimistic scientists (**Hilbert** completeness) used to think that everything could be solved through algorithms

- **Gödel's incompleteness theorems** (1931) and their computational consequences (**Kleen *Halting problem***), spoiled previous optimism

5.2 What can be done. Computability

- ***Halting problem*** states that there is no computer program that can correctly determine, given a program P as input, whether P would eventually halt when run with some given input…

- in other words, there is no algorithm capable of determining if other algorithm is about to finish

5.2 What can be done. Computability

- Related with these problems are those called:
  self-reference paradoxes

  Problems seeking a solution that is including the very same case of the definition…

  - Barber paradox: *The barber is a man in town who shaves all those, and only those men in town, who do not shave themselves?*

  - …is related to… Should *the set of all those sets that do not contain themselves*, contain itself?

Complexity

How to measure the ***time*** and/or the ***space*** (physical resources) that an algorithm would need for it to get its job done?

5.2 What can be done. Complexity

- Computational **Complexity** Theory studies the resources (space and time) algorithms demand to be executed

- Space: memory, disc, etc.
- Time: usually the most studied

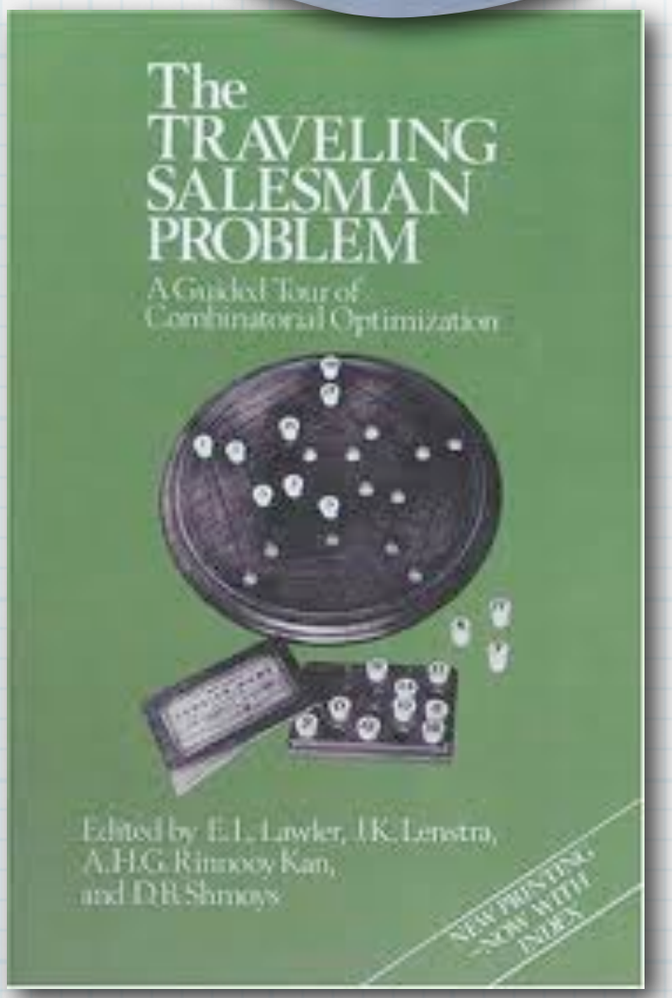
- From the Complexity point of view, problems can be:
  - **Tractable**: exists an *efficient* solution
  - **Intractable**: does not exist. For example,

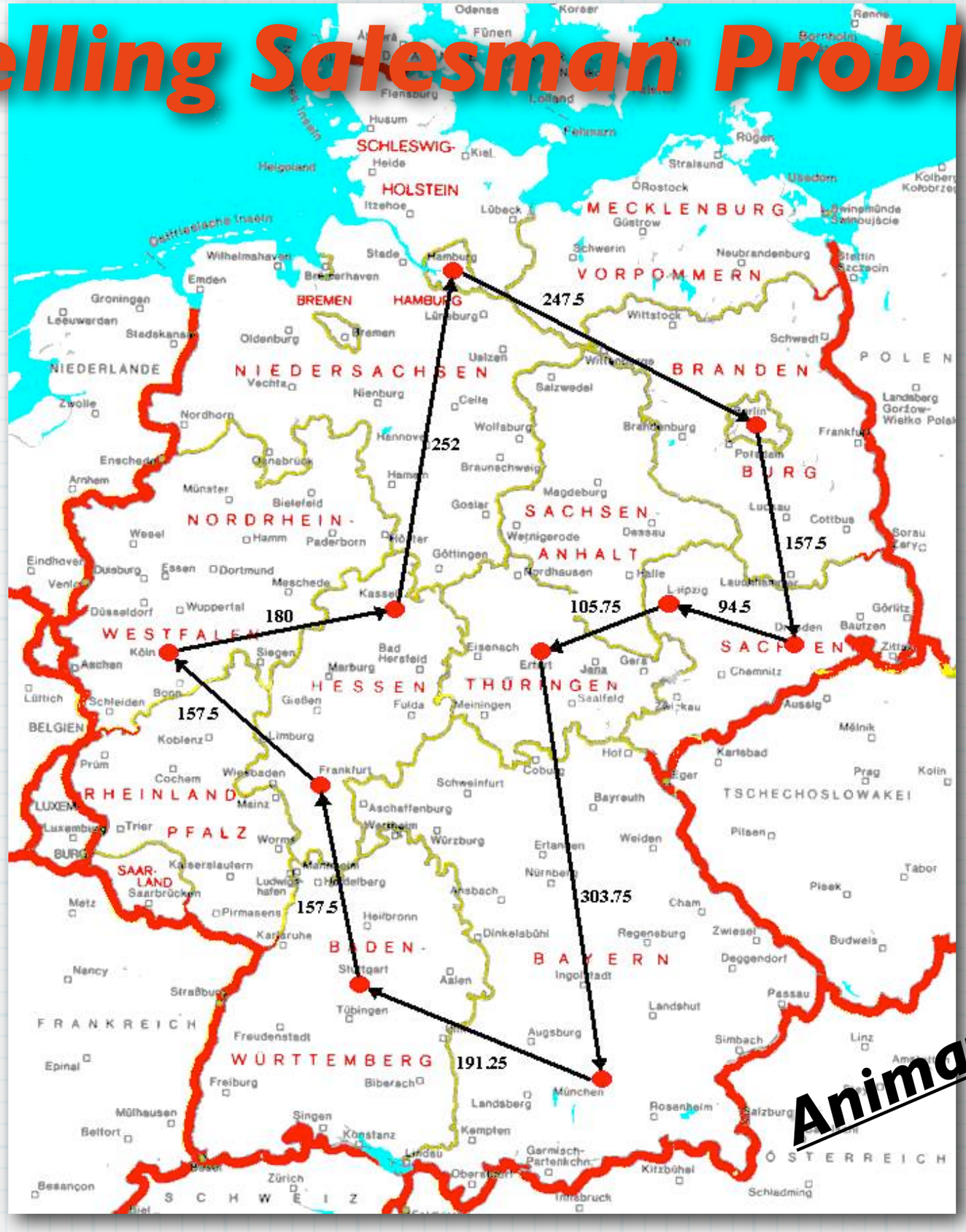
The Travelling Salesman Problem

- In the intractable problems what usually happens is you need to try every possible solution (a number of a size as $n!$), and then select the best among them

5.2 What can be done. Complexity

The Travelling Salesman Problem





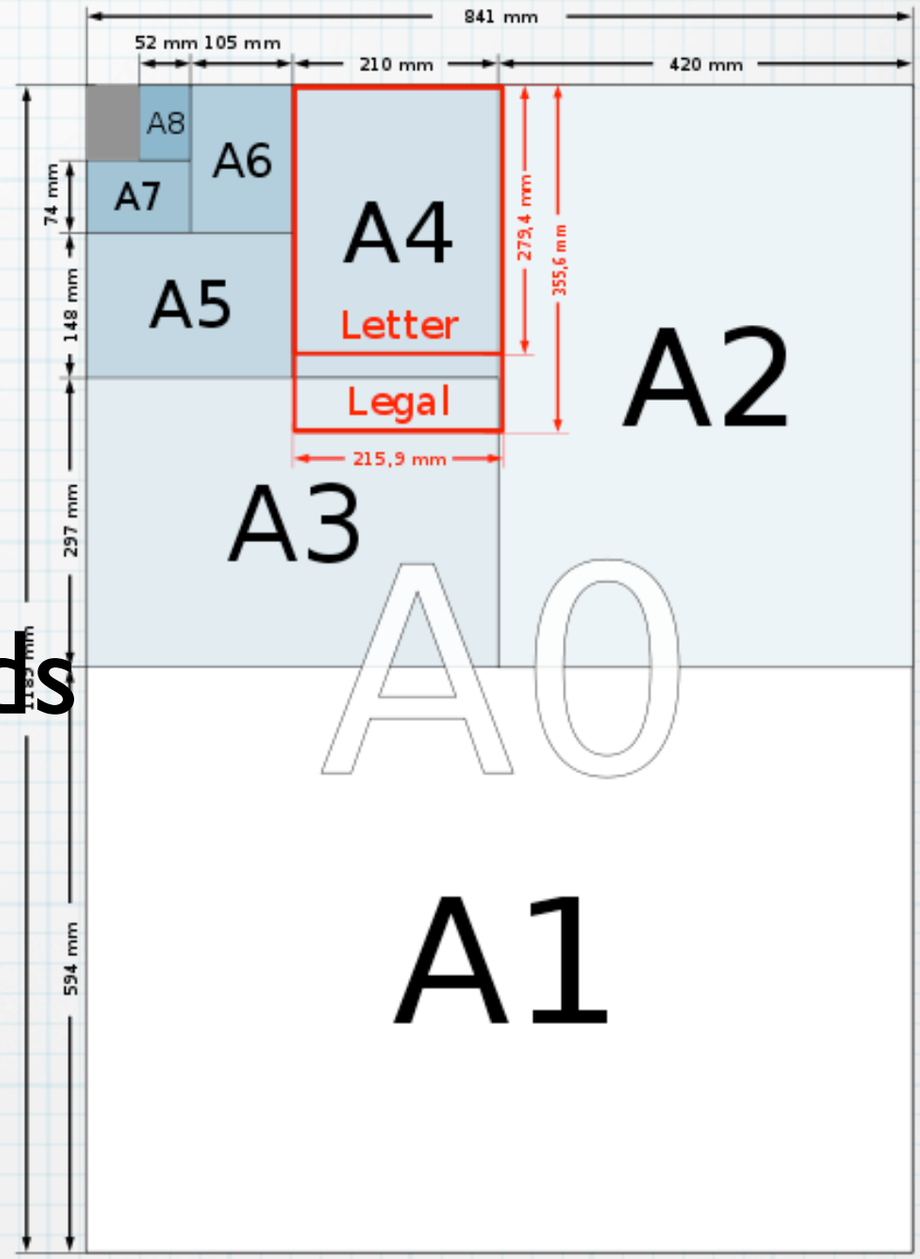
youtu.be/SC5CX8drAtU

5.2 What can be done. Complexity

- **Complexity of algorithms**: Measure the time a generic computer needs to achieve its goals
- Better algorithms last less or have slower growing as the size of the problem grows

5.2 What can be done. Complexity

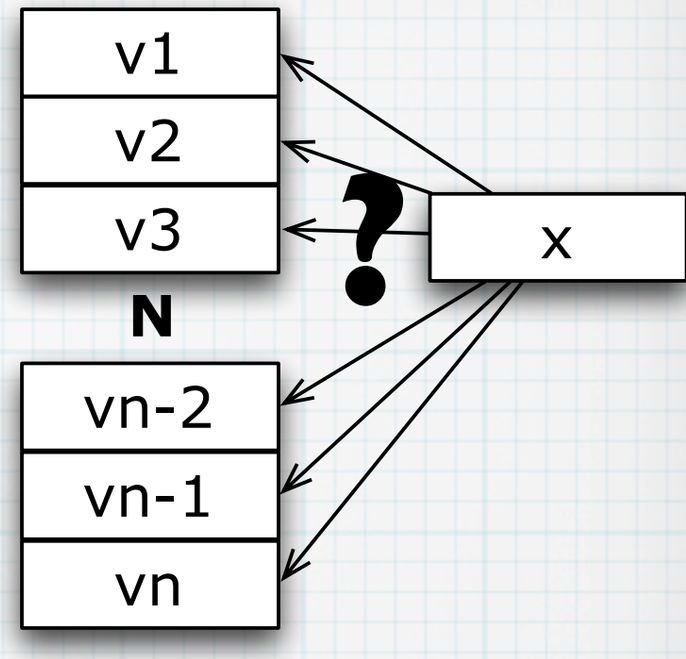
- **Complexity of algorithms** also assesses the RAM and disk space, in general, the physical resources an algorithm needs



5.2 What can be done. Complexity

- Since computers greatly vary in power,
time is not directly measured,
but the
total number of *abstract instructions* (***steps***)
algorithms take is what is finally accounted,

- … this quantity depends on the size/length/… of the data being processed



Time as function of steps

- Searching for a value among **N** values, requires at least **N** comparisons. This is an example of *linear complexity*,

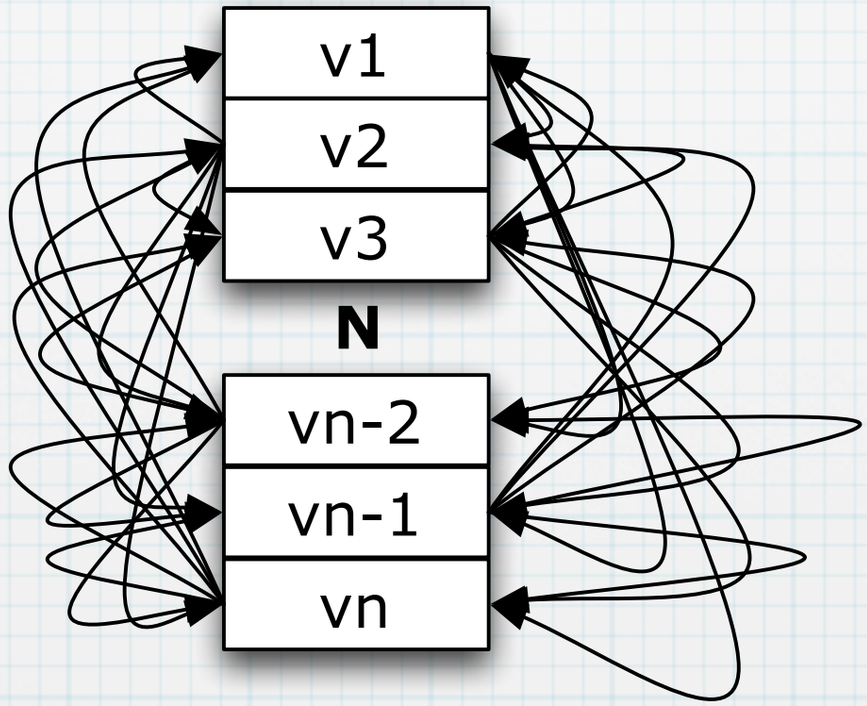
$$\mathbf{T}\text{ime}(N) = c\,N \quad \text{or} \quad \mathbf{T}(N) \propto N$$

- Here each step ***is*** a *comparison between to elements*

***Sorting*, number of steps**

- To sort **N** items in the most direct and simplest way, we need to compare each item with the all the others, then

$$\mathbf{T}\text{ime}(N) = c\,N \cdot N = c\,N^2 \quad \text{or} \quad \mathbf{T}(N) \propto N^2$$



$$\mathbf{T}(N) \propto N^2$$

5.2 What can be done. Complexity

- The dependency on the number of steps is also relative to the specific programming language, on the operating system, etc.

- A high level programming language may need many machine micro-steps to achieve each of its general commands

- It also depends on input data characteristics like order, values, etc

5.2 What can be done. Complexity

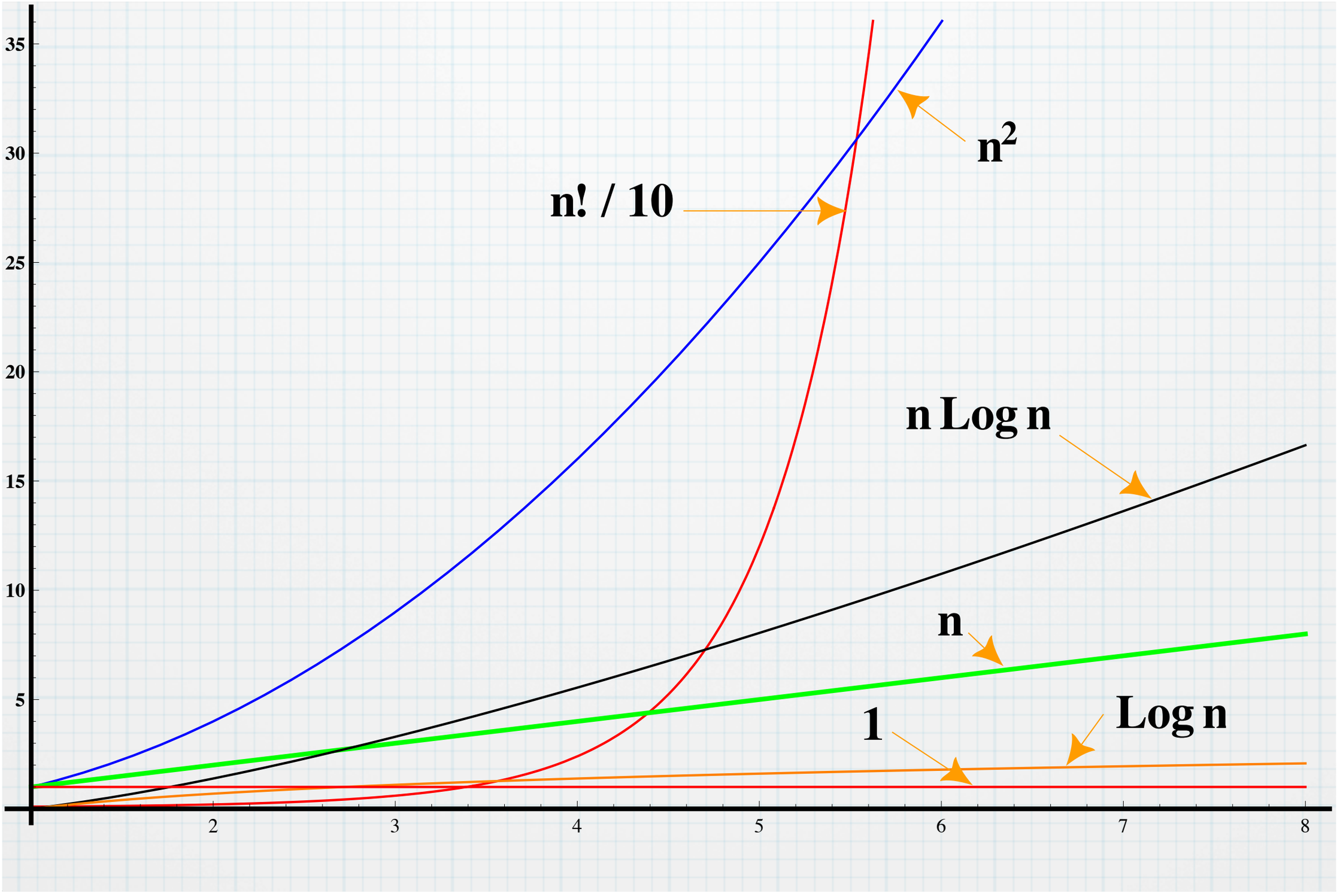
- We are not interested in knowing the exact number of steps for an algorithm
- To compute if a big number is prime or not, may be faster if the number is, for example, a power of 2
- Sorting an array is faster for most sorting algorithms when data are nearly sorted

5.2 What can be done. Complexity

- All in all, Complexity only tries to find a function asymptotically greater than T(N) as N grows, this is expressed as a bounding function, O(N)

- The complexity of the algorithm is O(N)

O(N) reference functions



5.3 How must be done. Correction

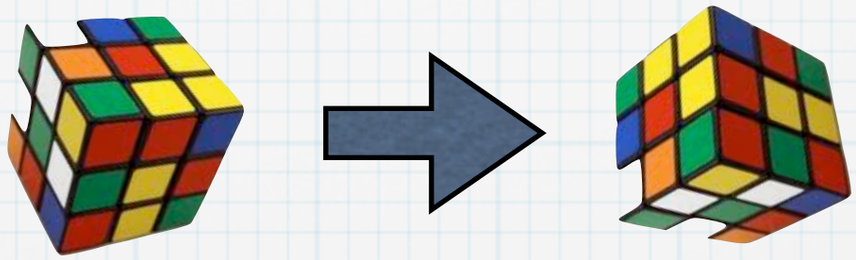
Correction of programs are achieved avoiding:

- **Lexical**, syntactical and semantical errors

  are easy to solve Ex.: y = *san*(x); y = *sin*(char);

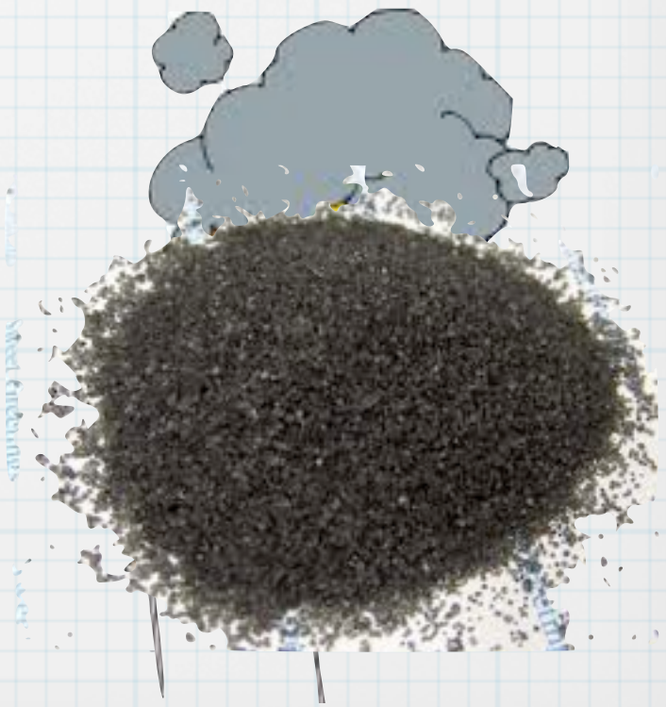
- **Logic** errors in the specification and design

  more costly



- **Execution** errors

  frequently depend on logical errors



5.3 How must be done. Correction

- To assure a program is correct there can be two ways:
  - **Practical:** experiments testing all possible inputs. Which is generally *impossible*, since may be a huge quantity of forms of inputs
  - **Theoretical:** formal verification of correctness…
    - *partial* –ending is supposed
    - *total* –ending is demonstrated
    - {P} S {Q}
      {preconditions} execution {postconditions}
      are logical asserts, like {x > 100} x^2 {x>10,000}
    - Verification must show P after S yields Q

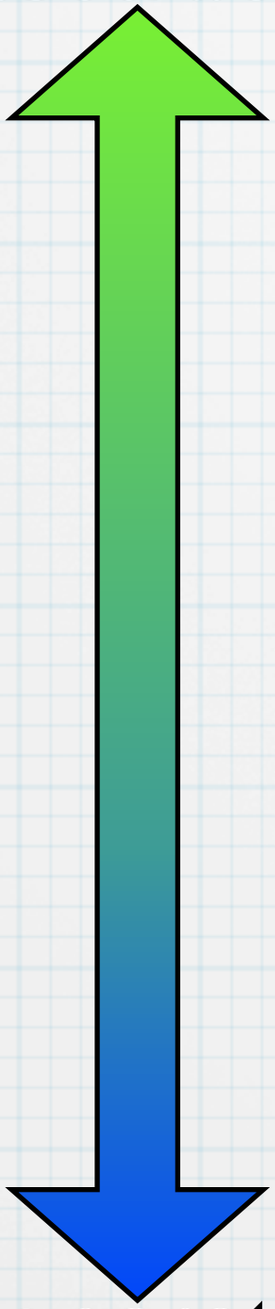
Part I. Introduction to Programming. Contents

6 Programming languages. Classification

- To classify programming languages we can consider
  - Abstraction **Level** (solve this polynomial)
  - **Purpose** (kind of problems that are aimed to solve)
  - **Paradigm** (or model of thinking on which it is based)

6 Programming languages

abstract



concrete machine details

Thought

Programming language

COMPILER C++

machine language 01010001

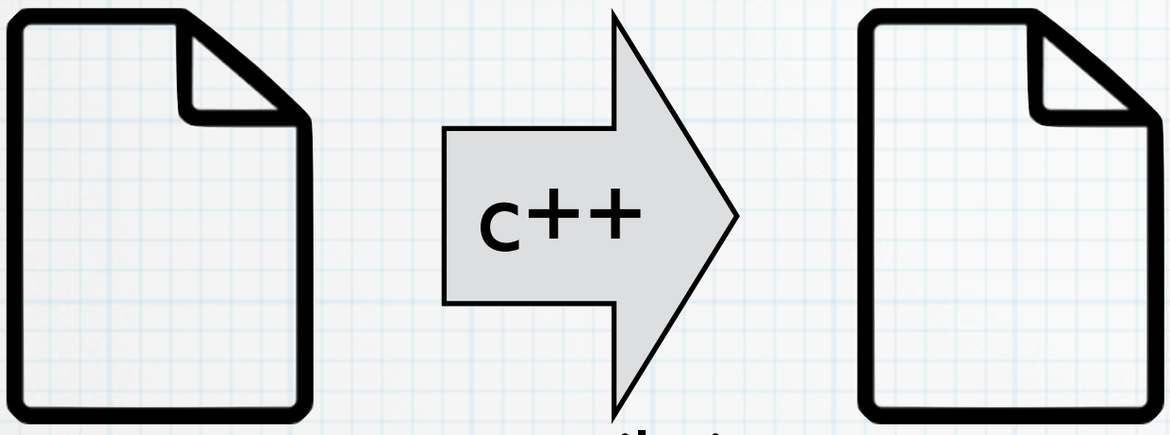
execution

6 Translators, compilers, interpreters

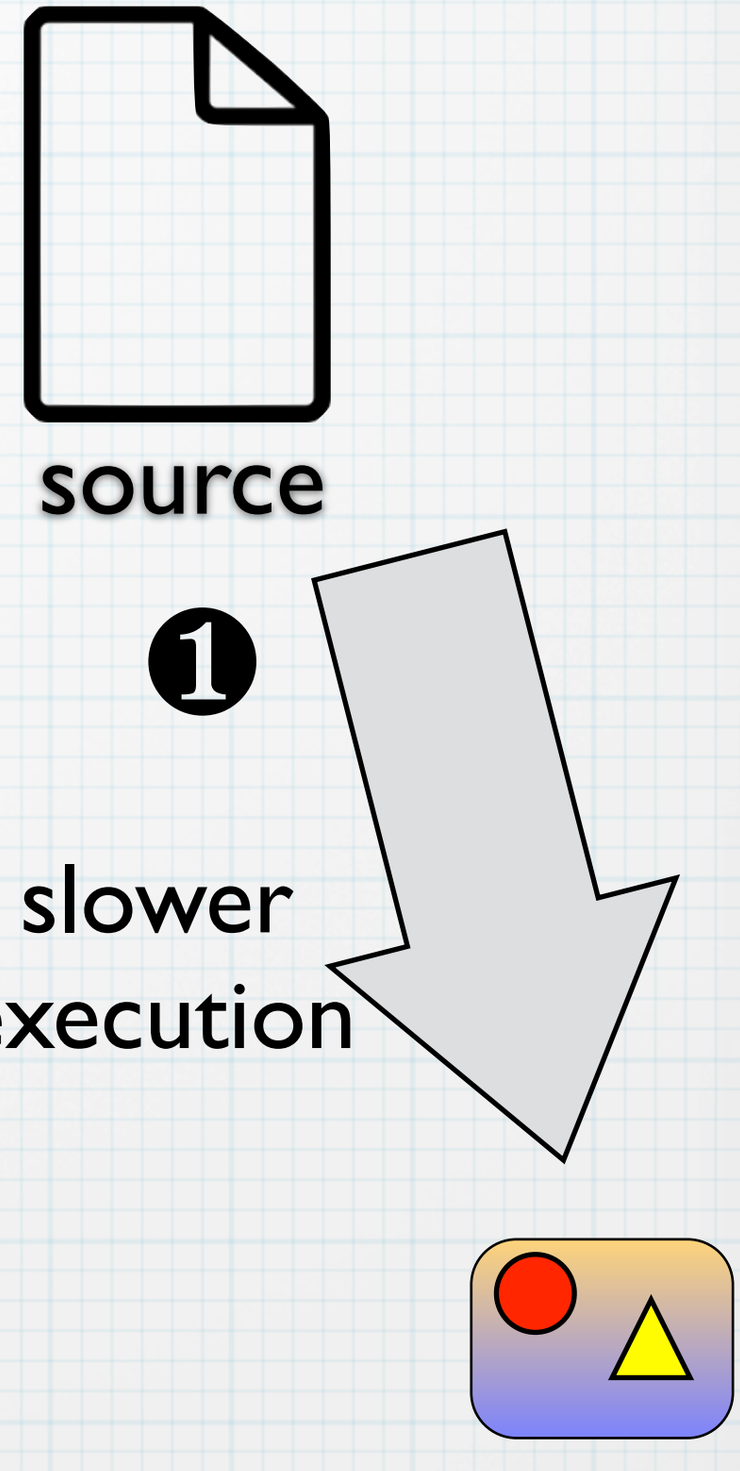
- **Translator** is a general converter, translates from one encoding to another
- **Compiler** converts from high to low level, it is specific to computer languages. It converts from our source.cpp to a machine language program file able to be repeatedly executed
- **Interpreter** acts *in-situ* executing high level, no conversion, as it reads the source, it executes it, without any other step

6 Compilers vs interpreters

Compilers



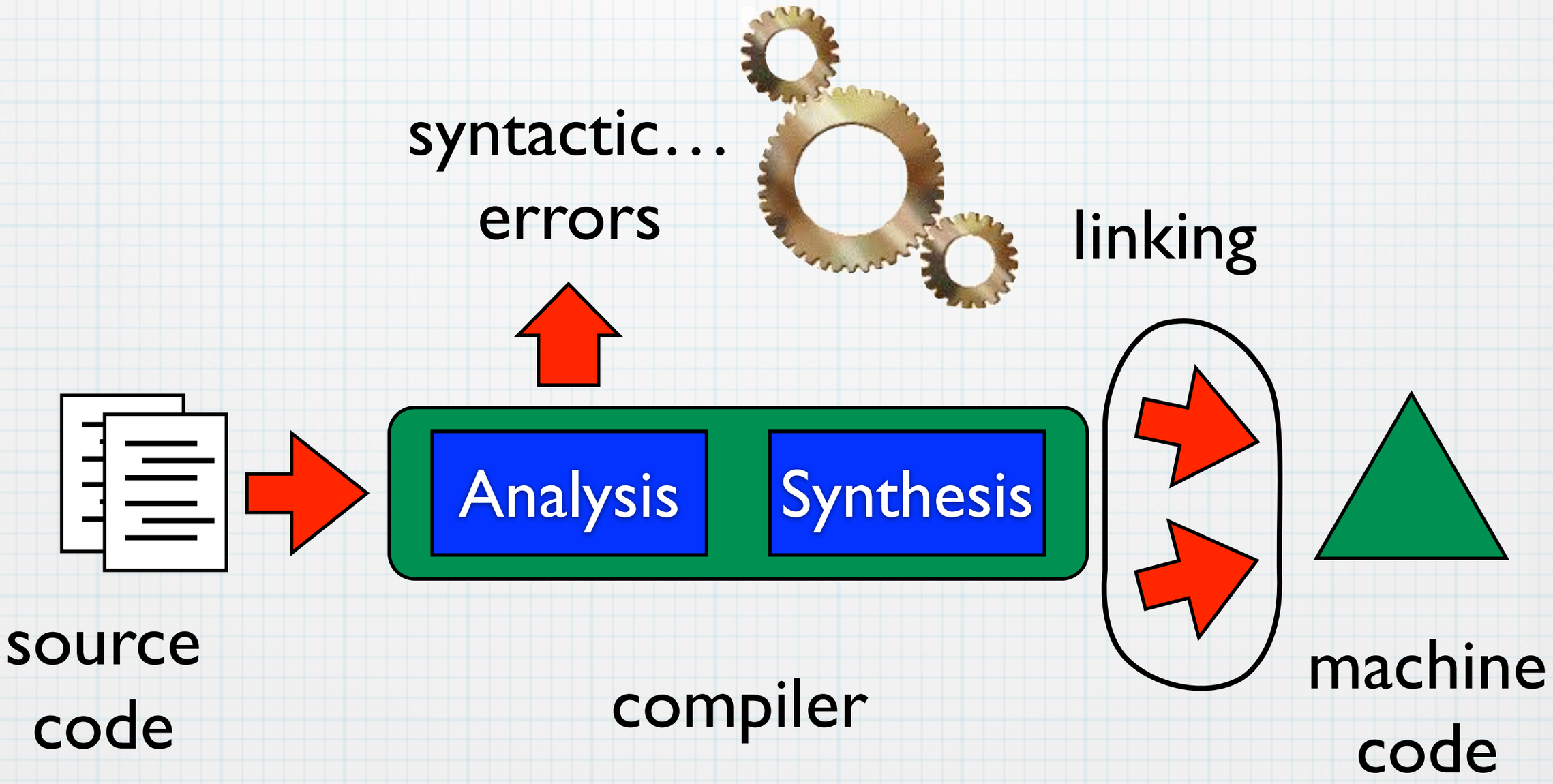
Interpreters



6 Translators, compilers, interpreters

- Compilers work in two **phases**:
  - **Analysis**: decodes and "understands" what we have written in our source.cpp
    - usually is where our syntactic and semantic mistakes show up
  - **Synthesis**: collect everything and build an efficient machine level code
- After compiling the ***linker*** joins together our output binary with those of the libraries we have referred to

6 Translators, compilers, interpreters



Part I. Introduction to Programming. Contents

7 The Computer System as a whole

- von Neumann architecture is the ***hardware*** organisation
- The Software organisation is based on big parts as:
  - Operating System (permanent layer)
  - System and Development Software
  - Tools (not produce documents)
  - Application software (produce documents)

7 Integrated Development Environments

- An Integrated Development Environment helps to develop and test (*debug*) our programs. A well known one is *Eclipse*. It mainly supply:
  - **Editor** (with **syntax colouring** and language oriented writing help system)
  - Underlying **Compiler**
  - ***Debugger*** (step execution; fixing of errors)

7 Integrated Development Environments

- A Terminal for UNIX/Linux commands and Editor may suffice
- The system must have installed a GNU C++ compiler version 4 or higher. All these if easily achieved in Ubuntu, OSX and Windows with Linux virtualized
- All of this will be detailed in the laboratory practices
- During the course you will need to have installed one of them MSCode, Xcode, etc. O perhaps, directly [edit+compile+execute] using not using a Graphical IDE but the universal and powerful Terminal as environment

The END