Methods in Computer Science education: Analysis 2021-22

Teaching Computational Thinking

Andrea Sterbini – sterbini@di.uniroma1.it

What are we doing here?

GOAL: How do we teach Computational Thinking?

WHY? (today)

- Define the Computation Thinking concepts
- Define the course structure and what will be your assignments

and HOW? (rest of the course)

- Analyse several learning environments/languages/programming styles
- Analyse example of CS curricula and of learning units
- Build learning units

WHY should we teach kids coding and C.T.?

1. To prepare new generations to new jobs? (?!?!?)

   What about AI-generated programs? What about programmers exploitation?

2. To ask kids to build stories in a different way than just writing?

   Story-telling as a creative way of creating and playing/moving characters

3. To vaccine youngsters against bad algorithms?

   Avoid being only program consumers and data producers

4. To empower everybody to be able to write her programs?

5. To introduce Computational Thinking <==

6. To introduce constructive didactics in any discipline <==

KEY effects of teaching Computational Thinking

Motivating students' interest

Robotics, Storytelling, Simulation, Social impact, Video-games, Embedded systems (see CSEDU: Design), CS Unplugged, Personal interests

Role playing and mental models of computation

Importance of Randomness in creativity, discovery, exploration

Simulation of Natural evolution / Artificial Intelligence

There are MANY <u>programming styles!</u>

Functional → filters and transformations

Procedural → drive a robot/agent

Declarative/logic → relations & rules

OOP → office metaphor

CS as the Science of "HOW TO DO/DESCRIBE/BUILD/SIMULATE"

A ‘BIT’ of History educational programming languages

When	Where	Language	Inspired by	Created by
1964	Darthmout	BASIC		[Kemeny & Kurtz]
1969	BBN	**Logo**	Lisp	[Feurzeig, Papert & Solomon]
1970	Zurigo	Pascal		[Wirth]
1981	Carnegie Mellon	Karel	Pascal	[Pattis]
1996	Apple/Disney HP/SAP	Squeak	Smalltalk	[Kay, Ingalls & Goldberg]
1996	Disney	e-Toys	Logo/Smalltalk	[Kay]
1999	NortWestern	**NetLogo**	Logo	[Wilensky]
2001		Guido van Robot	Python	[Howell]
2006	MIT	**Scratch**	Logo	[Resnick]
2010	India	**Kojo**	Scala	[Pant]
2014	Sacramento	**Flowgorithm**	Flowcharts	[Cook]

But also …

Alice (Java)

Blockly (visual)

- Code.org
- Appinventor

CiMPLE (C)

Kodu

Lego Mindstorms

Mama

Greenfoot (Java)

ToonTalk

Snap! (at Stanford)

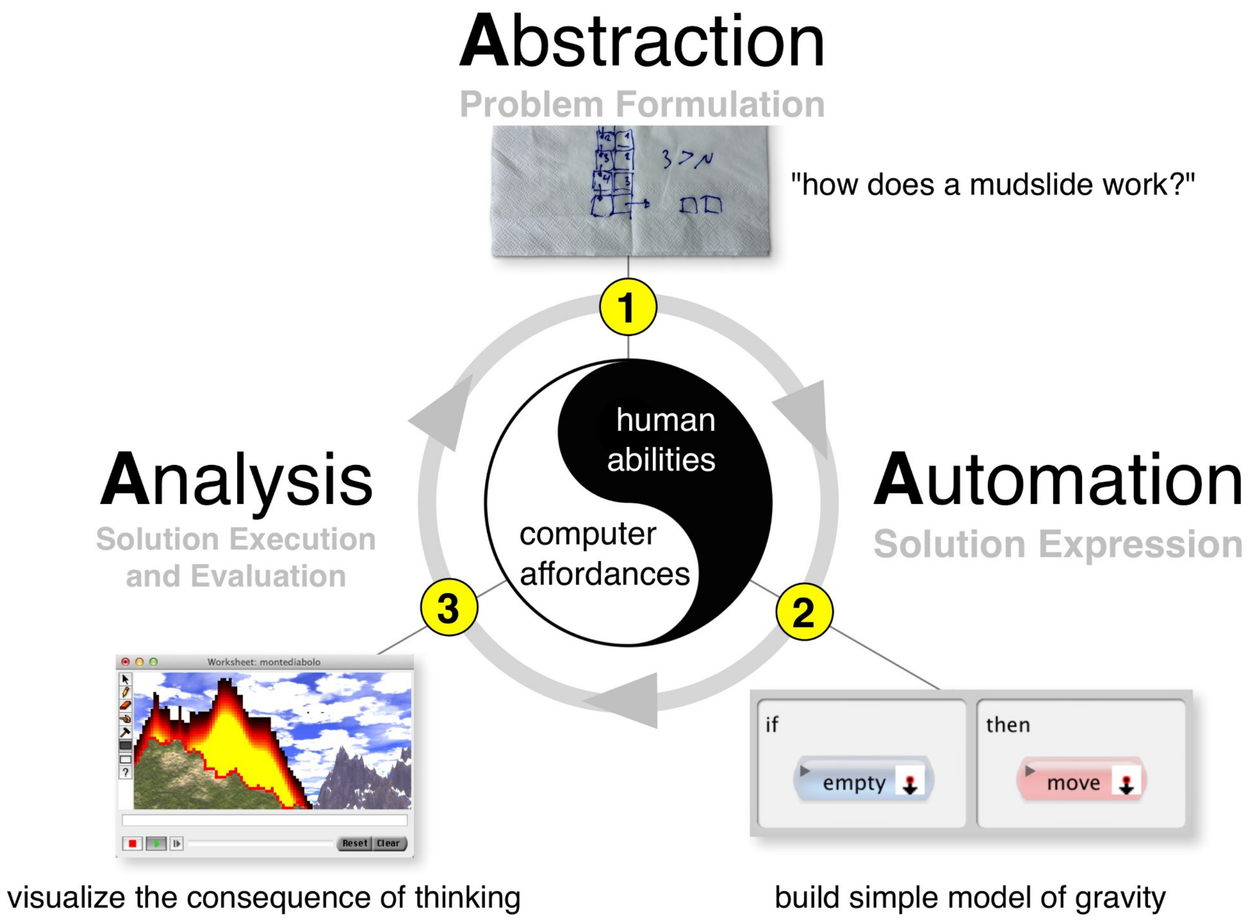
Stencyl

Prolog (text-based)

… and may others

(you can use the one you like)
Please suggest more!

WHAT is Computational Thinking? [Papert '80]



Abstraction

Analysis, representation

Automation

Planning steps
Define sub-problems,
and transformations

Analysis

Observation,
consequences,
evaluation, difference
w.r.t. predictions

Image by KaptainFire - Own work A. Repenning, A. Basawapatna, and N. Escherle, "Computational Thinking Tools," to appear at the IEEE Symposium on Visual Languages and Human-Centric Computing, Cambridge, UK, 2016., CC BY-SA 4.0, https://commons.wikimedia.org/w/index.php?curid=48453667

Computational Thinking
1) Abstraction

Abstraction of information/representation

Data representation, variables and memory, objects and attributes, types

Abstraction of process/control

Sequential algorithms, event-based programming, parallel programming, data flow, declarative programming, object oriented programming, functional programming

Abstraction of methodology / problem analysis

Top-down analysis, bottom-up analysis, declarative style, flow-based, pattern-matching rules, object orientation, functional, ...

Computational Thinking
2) Automation

Find a suitable <u>representation</u> for the <u>information</u>

<u>Split the problem</u> in small steps (or better said <u>“smaller problems”</u>)

Order them in one or more sequences/algorithms

Describe the data flowing between steps

Find a “suitable” implementation of the steps (algorithm)

- Within the **constrained resources** available (time, memory)

But also: (motivation for literate/well documented programming)

- Prepare for the **evolution/maintenance** of your solution
- **Keep track of the ideas** guiding your thoughts/analysis
- **Enable/empower others** to use your solution

Computational Thinking
3) Analysis of the execution

Prepare for observation

Choose good visualizations, show/spy intermediate data to expose inner details

Compare with expectations

Simulate the **algorithm in your head**, **predict the outcome** for simple cases, define test cases / examples

Diagnose discrepancies w.r.t. specification AND expectation

Find reasons for observed discrepancies, use assertions to early detect for anomalies, debug and observe the inner computation (variables **AND** flow)

==>> Better understand BOTH the problem AND the computer

The **problem description/specification** could be challenging to fully grasp
The **programming** language, functions, libraries can be tricky to master

BUT: What about the Social impact of C.T.?

C.T. could be seen as too much focused on the C.T. process

Abstraction / Automation / Analysis

A critique moved to C.T.:

little analysis of the impact on other fields

Reuse and modularity, analogy, <u>social impact</u>

For this reason (and others) we will design <u>interdisciplinary</u> units

And we must give a lot of attention to the program “life”
and to the data required, managed, deduced

Why one should learn C.T.?

Pro:

Computer Science is the Science of **HOW (to represent, to compute, to solve)**

You will see other fields/subjects (Society, Music, Language, Art, Medicine . . .) with a different analytic / creative eye

Society is more and more computer-based, therefore knowing how to write/understand programs makes you **less dependent** on others

You can **explore (virtually and physically) new ideas** at relatively low cost

Even if you WILL NOT program, you will **understand the possibilities** and you will be able to **describe what you want** to be programmed/created

Con:

Shabby/good-enough solutions trick you into **false understanding and lazy methodology**

The **social impact** of a program or of its data could be way bigger than you think

False assumptions we heard about Computers ...

You just need to know how to USE a computer (?!?)

Computers are FAST

- BUT DUMB!!! Limited instructions BUT bloody fast CPUs and intelligent algorithms

Computers are FLEXIBLE and MULTI-PURPOSE

- BUT RIGID and UNFORGIVING :-) There are soooooo many details to be aware of (declarations, initializations, scope, arguments, program termination, syntax, errors ...)

Computers SAVE YOUR TIME, Programming is EASY (!?! WTF !?!)

- BUT programming is TIME-CONSUMING, you must be EFFICIENT and PERSISTENT:
  - When you code: (good IDEs, good documentation, easy programming languages, ..., GOOD METHODOLOGY)
  - When you run (efficient algorithms, special data structures, ...)
  - When you fix YOUR (or other's) mistakes (good documentation, good tests)

Computer can store HUGE amount of data

- BUT RAM memory space is limited. Virtual Memory helps but SLOOOOWS DOWN EVERYTHING

What new concepts are introduced because of Computers? (methodology level)

Problem solution by <u>reduction to smaller problems</u>

Algorithm as a sequence of actions

(but see also <u>declarative</u> / <u>parallel</u> / <u>data-flow</u> / <u>rule-based</u> programming or ... <u>neural networks!</u>)

Data representation

Algorithms must manage some **meaningful representation** of information

Constrained execution! (time, memory)

Simulation as tool to explore the impossible ("What if?")

Explore multiple consequences in a virtual world with new rules

Empowerment and collaboration of the individual in the society

Open-data, Open-formats and Open-source development enable the single to **collaborate with others** and tackle global issues

Social issues of the information you receive/derive

Information as a good to be sold/exchanged. **Sensitive data**.

Motivation, in school, could be a huge problem

Teaching programming to <u>university students</u> is easier (!?!?!)

- <u>They chose it</u>, and we (try to) go deep in many interesting ways

Some <u>high school</u> students didn't choose the topic, but could be motivated by raising their interests with <u>concrete interesting problems</u>

- Robotics, Embedded systems (see CS-edu:Design), Storytelling, Simulation, Social impact, Video games, Personal interests, Local issues, Mobile apps

<u>Role playing</u> can make C.T. concepts very clear <u>in a playful way</u> to younger students to understand what a computer is/does

- They could either pose as the "programmed agent" or be the "programmer god"

"CS Unplugged" activities show C.T. methods without a PC

- Appealing for very very young students

What new concepts are introduced because of Computers? (computer specific)

Program = Precise algorithmic definition of a solution

STATE changing through time (THE main difference w.r.t. Math)

Information representation/encoding, data types (analogy with Physics?)

Names vs memory (HUGE misunderstandings arise here)

Functions, arguments, return values

Side-effects! (and bloody global variables)

Language syntax (bloody parentheses and semicolons)

Objects, attributes (and again, changing state)

Methods as object's actions/abilities, the office metaphor

Control structures (loops/repetition, conditions)

How to analyse and build a program?

Top-down analysis

- Define input/output data representation
- Write an high-level description of the problem, divided in subproblems
- Implement the algorithm by defining mock functions for each step, mimicking their I/O
- If needed:
  - define the additional intermediate data passed between steps
  - add the initial data definition and initialization
- Test if the logic is correct
- Repeat the analysis/implementation on each high-level step/function so defined
- When the steps are sufficiently detailed and similar to the programming language constructs, implement the details of the actual program

Be aware that

- Global variables → side-effects hidden from functions definition and usage
- Poor control structures and poor logic can produce inefficient/endless computations

Other analysis methodologies

Object-oriented

Define classes of objects responding to requests and interacting with each other. Try to reuse/standardize behaviours/definitions to simplify interoperability of objects and algorithms. Find common procedures but allow for exceptions.

Event-based (GUI, e.g. see Scratch, Snap, AppInventor)

Describe how a collective set of objects should **react to external events**

Declarative/Logic-based (Prolog)

Describe **relations** among data and how more complex **properties can be derived** from simpler ones. **Let the system find a solution** plan.

Bottom-up

Start from small reusable data manipulations and build more complex ones.

How other subjects can benefit from Computer Science methods?

Exploration of laws and rules by modelling and simulation

Physics, Combinatorics, Chemistry, Geometry, ...

Exploration of creativity by building computational models

Language generation and analysis, Music generation, ...

Algorithmic description of problems/solutions or of rules

Math simplification, Language analysis

Learning a methodology to analyse problems

Data representation: a way to capture regularity and exceptions

Randomness: a tool to explore creativity (and mimic intelligence)

Simulation of Darwin's evolution, creation of artistic paintings/3D scenery

What approaches can make easier learning C.T.?

Syntax is considered one main initial problem for younger kids

- We could completely **remove the syntax** by using visual programming
  - Joining **snap-on blocks** (Blockly, Scratch, Snap! and similar)
  - Drawing **flow charts** to describe the control flow (Flowgorithm)
  - Drawing **data-flows** to describe the data flow (LabView and similar)
  - Editing **multiple agent properties/predefined behaviors** (GameMaker, Alice, ...)
- Or **simplify the syntax** to make the programs easier to read/write
  - Logo, Smalltalk, Python, Ruby, Scala, (Prolog), Occam, . . .

Helping the student to build a mental model of what happens

- Visualizations of the **inner program status** (variables, execution, debug)
- Visualization of **external effects** (simulated agents moving around, robots)

What Learning environments could be used?

In the rest of the course we will:

- Analyse environments/languages built for learning how to program
  - Visual-based: Snap!, Scratch, Blockly, OpenRoberta, AppInventor …
  - Logo-based: NetLogo, LibreLogo
  - Scala-based: Kojo
  - Logic-based: Prolog
  - Flowchart-based: Flowgorithm
  - **Data-flow based**: LabView, …
- We will **build an example** learning unit within the environment/language
- We will find and **analyse learning experiences** from around the world
- You will **suggest/discuss/plan new learning units**
- You will **build and present the learning units** designed

How others are teaching C.T. around the world?

Visual programming

Scratch Blockly Snap! AppInventor OpenRoberta

Programmareilfuturo.it code.org …

Commercial

Microsoft Minecraft Education edition	education.minecraft.net
Apple Swift Playgrounds (on iTune)	
Wolfram	computationalthinking.org

Less knowns approaches

Flowgorithm, LabView, NetLogo, Alice …

Course prerequisites

You MUST be fluent in at least <u>two</u> programming languages

Python? C/C++? Java? Pascal? Ruby? Lua?

Prolog? Scala? JavaScript? Assembly? Go? ???

You MUST be fluent in at least <u>two</u> programming paradigms/styles

Procedural? Object Oriented?

Declarative/logic? Functional?

Data-flow? ???

Please fill the on-line questionnaire

http://bit.ly/CSedu-q1

Course methodology

The course is very hands-on, we will

- Use **many** learning environments, visual or textual
- **Analyse** their strengths/weaknesses w.r.t. learning Computational Thinking
- **Analyse** learning units built by others (including your peers of AA18-21)
- **Design and Build** complete functioning learning units

We focus ONLY on <u>interdisciplinary</u> learning units

- To apply the **Computational Thinking methodology**
- To show that programming helps **understanding/exploring** the problem to be solved
- <u>And thus to **constructively solve** the interdisciplinary task</u>

<u>Comments/suggestions/improvements/critiques are WELCOME</u>

Course assessment

You will build 3 new interdisciplinary learning units in 3 different learning environments/systems of your choice (2/3rd of the grade)

At most 2 LU can be made with block-based systems

You can work either alone or in small groups (max 2). Groups are expected to produce more complex learning units. The group work done should be clearly split among the participants ("who did what?")

Learning unit presentation and discussion (1/3rd of the grade)

You will present and discuss with the rest of the class your learning units, describing motivations, methodologies, features, experienced problems, possible problems for application in class and proposed solutions

"Net-borrowed" learning units must show what is your contribution (but, anyway, I will ask for improvements / heavy modifications)

Schedule of the course

7 Lessons

End of March: propose/discuss/present your 1st Learning Unit

7 Lessons

End of April: propose/discuss/present your 2nd LU

7 Lessons

End of May/exam: propose/discuss/present your 3rd LU

Exam

What's important in your Learning Units 1) WRT the interdisciplinary subject

MUST BE interdisciplinary = solve problems in non-CS subjects (it CANNOT BE a programming game or quiz)

Deliverable: 1 PDF report + 2 programs

PDF describing the interdisciplinary topic and the Learning Unit

- Prerequisites, motivation and placement in the course/school
- Describe the organization of the lesson, the topic, the task to be solved

REMEMBER: You are the expert

- Choose the topic wisely and **study it very well**

2) WRT Computational Thinking/Implementation

The implementation MUST use some <u>data structure</u> declaratively

Describe <u>Prerequisites and Placement</u> wrt to programming knowledge

Describe the data available, the data computed, the algorithms/interactions, the libraries given to the students

Explain WHY did you chose that development system?

- Try to “hero” (use in a prominent way) the system’s best features

Assessment grid describing how you will grade the programs

- <u>Build an example of Minimal (6/10) and Maximal (10/10) implementations</u>

REMEMBER: You are the expert

- Show **beautiful** well-modularized and documented code

Course site (on twiki)

Chat about your experiences

Fill the on-line questionnaire

http://bit.ly/CSedu-q1

(it takes just 2 minutes)

Send me your Telegram handles (just for emergency comms.)

sterbini@di.uniroma1.it (for comments/suggestions)