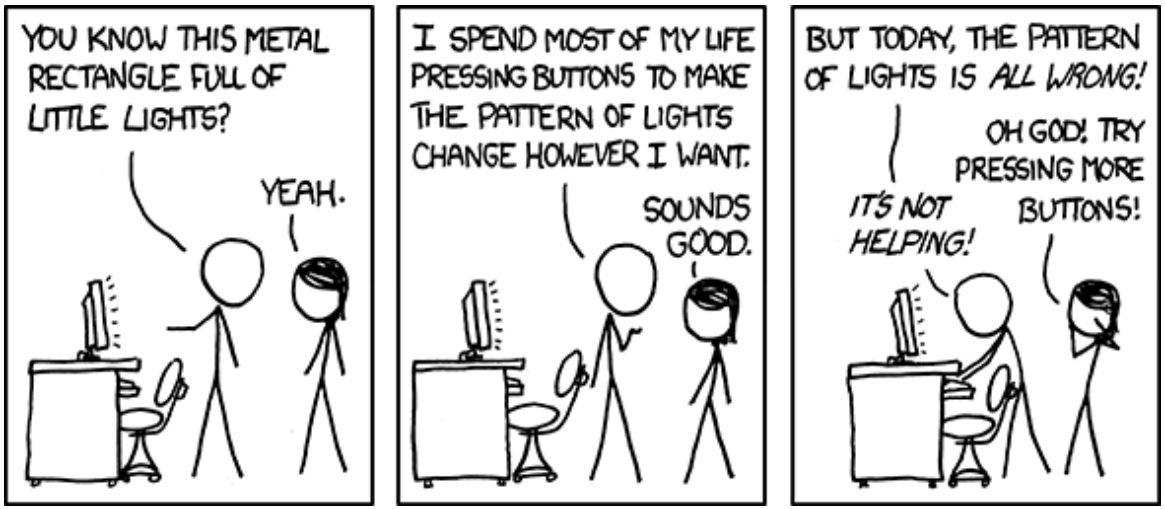
(about) Algorithms I: Correctness

Is it right... for every input?

Pedro Ribeiro

DCC/FCUP

2026/2027



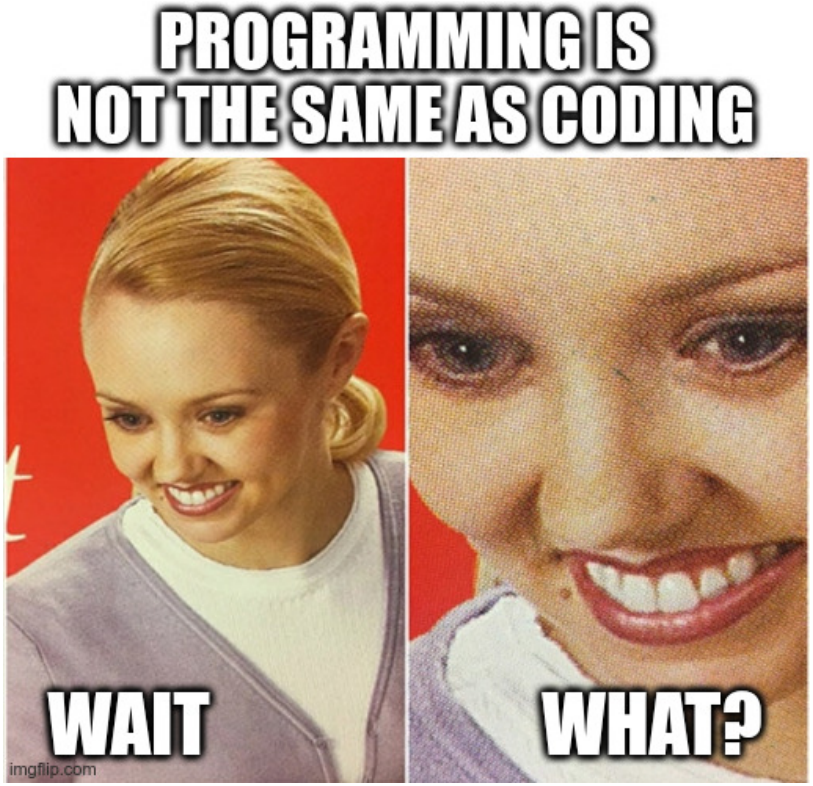
What are we doing in this course?

Our course

- What is the name of this course?



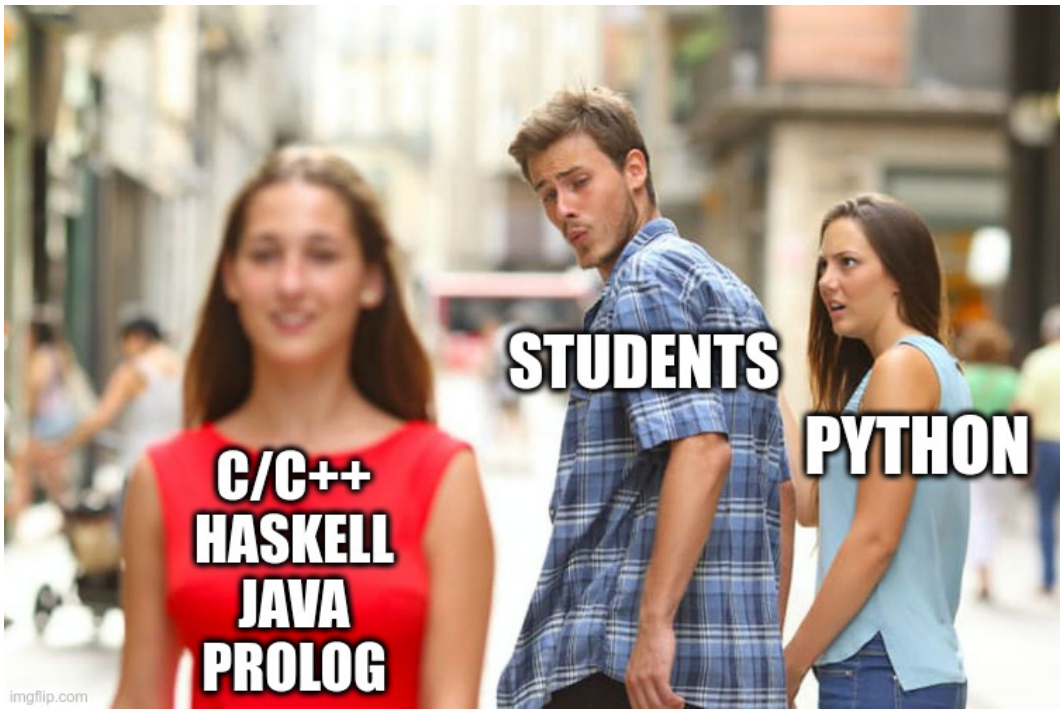
Programming vs Coding



- **Programming** is the mental process of **thinking** up instructions to give to a "machine" (like a computer).
- **Coding** is the process of transforming those ideas into a **written language** that a computer can understand.

Starting with Python

- In the coming semesters you **will** learn other languages



For example:

- **C**: Imperative Programming
- **Java**: Data Structures
- **Haskell**: Functional Programming
- **Prolog**: Logic Programming
- **C/C++ or Java**: Design and Analysis of Algorithms
- ...

- **But the programming concepts will remain!**

Same idea, many languages

Sum all the elements of a list: **go through the elements, accumulating a total**

Python

```python
def mysum(lst):
    total = 0
    for x in lst:
        total += x
    return total
```

C

```c
int mysum(int v[], int n) {
  int total = 0;
  for (int i = 0; i < n; i++)
    total += v[i];
  return total;
}
```

Java

```java
static int mysum(int[] v) {
  int total = 0;
  for (int x : v) total += x;
  return total;
}
```

Haskell

```haskell
mysum []     = 0
mysum (x:xs) = x + mysum xs
```

Prolog

```prolog
mysum([], 0).
mysum([X|Xs], T) :-
    mysum(Xs, T1), T is T1 + X.
```

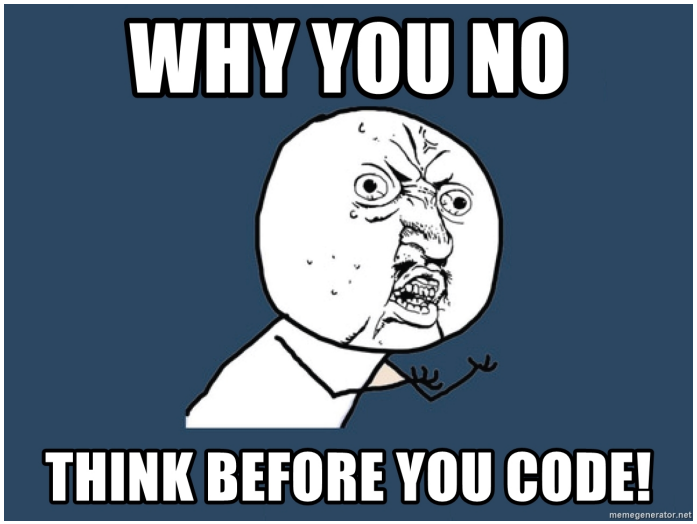
The **syntax** changes...
the **idea** does not!

Today we talk about ideas, not about languages

- Today we will use Python to **show** things...
- ...but almost nothing today is **about** Python

- Every mistake we will see could be made in **any** language
  (because the mistake is in the *thinking*, not in the *typing*)

- Today's goal: start to **think computationally**
  (going from "I can implement this" to "is this the right thing to implement?")



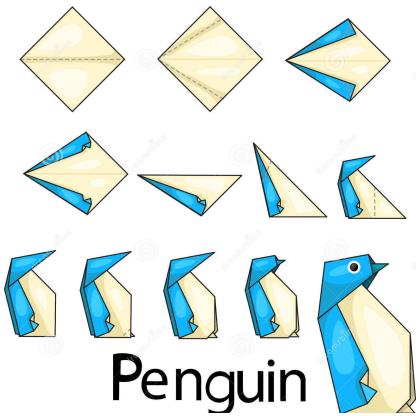
Algorithms as the core ideas

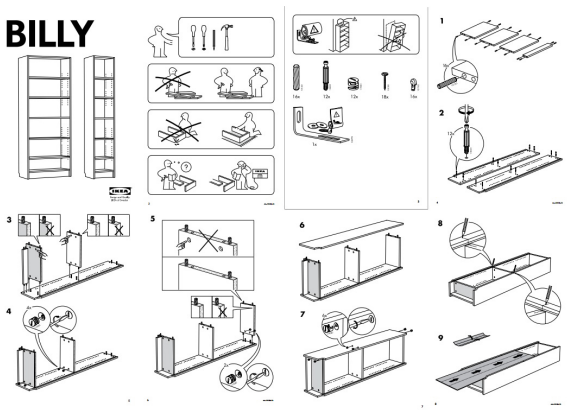
What is an algorithm?

A set of executable **instructions** to solve a **problem**

- The problem is the **motivation** for the algorithm
- The instructions need to be **executable**
- There are generally **several algorithms** for the same problem [How to choose?]

- **Representation**: description of the instructions clear enough for its "audience"






```
def fib(n):
    if n < 2:
        return n
    return fib(n-1) + fib(n-2)

print(fib(10))
```

What is an algorithm?

Computer Science version

- The algorithms are the **ideas** behind programs
  They are independent from programming language, machine, ...
- An algorithm solves a **problem**
- The problem is characterized by the description of its **input** and **output**

A classic example:

Sorting Problem

Input: a sequence $\langle a_1, a_2, \ldots, a_n \rangle$ of n numbers
Output: a permutation of the numbers $\langle a_1', a_2', \ldots, a_n' \rangle$ such that $a_1' \leq a_2' \leq \ldots \leq a_n'$

Example for the Sorting Problem

Input: 6 3 7 9 2 4
Output: 2 3 4 6 7 9

Properties we want in an algorithm

Correctness **[today]**

It should solve correctly **all instances** of the problem

Efficiency **[next class]**

(**Time** and **Memory**) performance has to be adequate

Correctness

Correctness of an algorithm

- **Instance**: A concrete example of a valid input
- A correct algorithm solves **all possible instances**
  Examples for sorting: repeated numbers, already sorted sequences, ...
- It is not always easy to **prove** that an algorithm is correct...
- ...and often it is not even **obvious** whether it is correct or not

Three ways of being wrong

1. **Wrong about the inputs**
   we know what we are doing... but not which instances can appear, or what the answer should be for them
2. **Wrong code**
   the idea is fine... but the implementation has a bug
3. **Wrong idea**
   the algorithm itself fails for some instances

In all three cases, the cure is the same:
think about the difficult inputs!

1. Wrong about the inputs

The warmest day of January

- We have the maximum temperature of every day in January
  (a list of integers, in °C)
- We want to know the **warmest** day: the maximum of the list

"Easy! Keep the best so far, update when something bigger appears"

```
def warmest(temps):
    best = 0
    for t in temps:
        if t > best:
            best = t
    return best
```

```
>>> warmest([14, 12, 16, 11, 15])      # Porto
16
```

Looks correct... is it?

The warmest day of January

- Now let's go to Oymyakon (Siberia), one of the coldest inhabited places on Earth

```
>>> warmest([-38, -41, -35, -44, -39])      # Oymyakon
0
```

It was never 0°C in Oymyakon in January!

- The **idea** was fine, the **code** does what we wanted...
- ...but we assumed something about the inputs that was **not true** (temperatures can be negative)
- Fix: our first guess should be an actual element of the list

```
    best = temps[0]
```

- And what if the list is **empty**? (the problem statement must say!)

Who gets the silver medal?

- We want the score that gets **silver**: the second largest

```
def silver(scores):
    return sorted(scores)[-2]

>>> silver([3, 8, 5, 7])
7
```

```
>>> silver([5, 5, 3])
5
```



Photo: Bob Ramsak, CC BY-SA 4.0

- Is that right? Wait... what **should** the answer be? 5? 3? Nobody?
- Tokyo 2020, high jump: Barshim and Tamberi **shared the gold medal**... ...and **no silver medal** was awarded! (the next one got bronze)

Before asking "is my code right?", ask "what **is** the right answer?"
The problem statement must say what happens with **ties**

Lesson #1: know your instances

- What **values** can appear?
  negative? zero? huge?
- What **sizes**?
  empty? only one element?
- Can there be **repetitions**? **ties**?
- What does the problem statement **actually** say?
  which inputs are valid... and what is the right answer for each?



2. Wrong code

Cleaning sensor data

- A sensor records one reading per minute
- When it fails, it records a **negative error code**
  (e.g. -1, -3, -7, ... depending on the type of failure)
  (and when sensors fail, they tend to fail for a while...)
- We want to **remove all the negative values** from the list
  (changing the list itself: other parts of the program use it)

The idea

Go through the list and remove each negative element we find

Simple, right? The idea is correct. Let's code it.



LIVE CODING SESSION

Cleaning sensor data

live coding recap

```
def clean(readings):
    for x in readings:
        if x < 0:
            readings.remove(x)
    return readings
```

```
>>> clean([12, -3, 15, -1, 14])
[12, 15, 14]
```

Works!

```
>>> clean([12, -3, 15, -1, -7, 14])
[12, 15, -7, 14]
```

A -7 survived!

- When we remove an element, everything after it shifts one position to the left...
- ...and the `for` moves on to the next position, **skipping** an element
- It only fails when there are **two consecutive** negative values

Cleaning sensor data

live coding recap

Another attempt, another failure

```
def clean(readings):
    for i in range(len(readings)):
        if readings[i] < 0:
            del readings[i]       # same as readings.pop(i)
    return readings
```

```
>>> clean([12, -3, 15, -1, 14])
IndexError: list index out of range
```

- Same bad idea (changing the list while traversing it), different symptom

- Which is worse?
  - a program that **crashes**
  - a program that **silently gives a wrong answer**

(the silent one: nobody notices until it is too late!)

Cleaning sensor data

A correct version

live coding recap

Keep the idea, fix the code: only move forward when nothing was removed

```
def clean(readings):
    i = 0
    while i < len(readings):
        if readings[i] < 0:
            del readings[i]
        else:
            i += 1
    return readings
```

```
>>> clean([12, -3, 15, -1, -7, 14])
[12, 15, 14]
>>> clean([-1, -3, -7])
[]
```

Lesson #2: the idea was right, the code was not

And it was a **specific kind of input** that exposed the bug

(if we had only tested the "nice" example, we would never know)

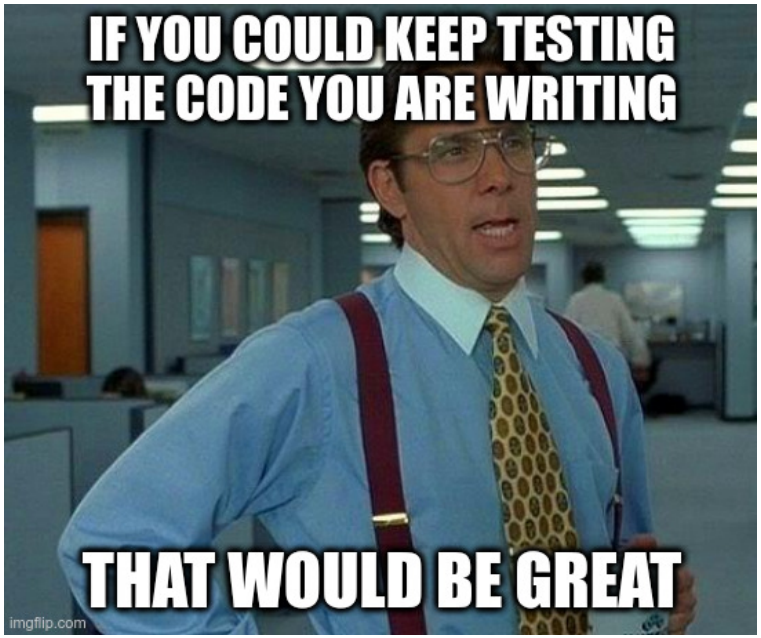
(if a **new** list was allowed, we could also build one with only the valid readings... which would even be faster: more on Friday!)

Testing

Never throw a test away

Every input that broke your code becomes a test you **keep**:

```
# typical
print(clean([12, -3, 15]) == [12, 15])
# the one that broke us
print(clean([12, -1, -7, 14]) == [12, 14])
# all negative
print(clean([-1, -3]) == [])
# empty
print(clean([]) == [])
```



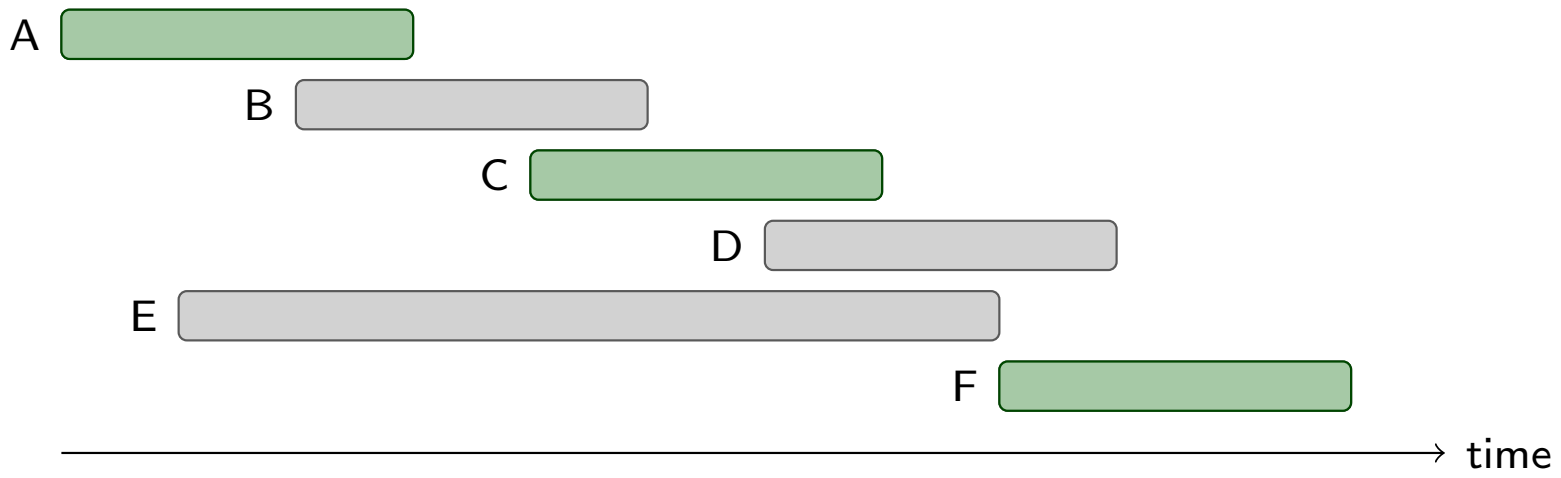
Run them again after **every** change: all must say `True`

(Python even has **`assert`** for this: silent when it is true, an error when it is not)

3. Wrong idea

The festival problem

- A music festival has many concerts, each with a **start** and an **end** time
- You want to watch the **largest possible number** of concerts (complete concerts, and you cannot be in two places at once)



A, C, F: 3 concerts (can you do better?)

- With hundreds of concerts we need an **algorithm**. Ideas?

The festival problem: a greedy approach

Greedy algorithm

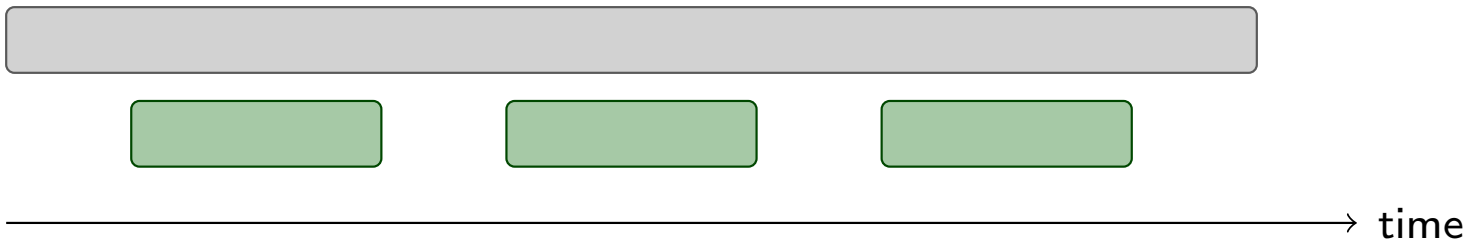
Repeatedly pick the "best" concert according to some **rule**, discard the ones that clash with it, and repeat

- Which rule?
- A natural first try: the one that **starts first**
  (the first concert of the day, then the next one I can still reach, ...)

Let's try to **break** it!

(to the blackboard...)

Greedy idea #1: the one that starts first



Greedy: picks the long concert $\rightarrow$ **1** concert

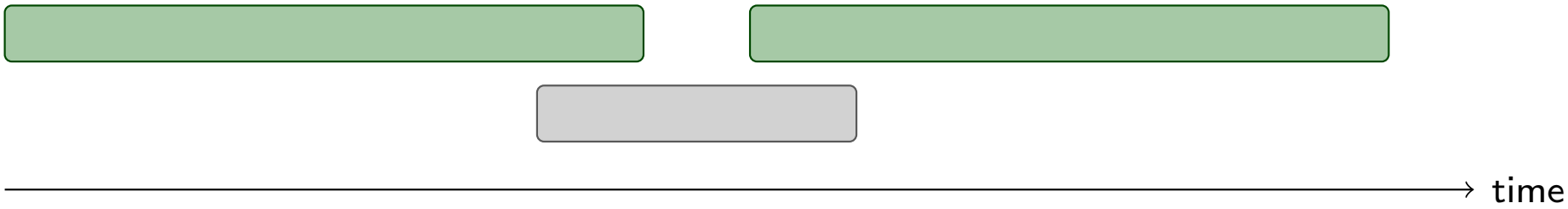
Best: the three short ones $\rightarrow$ **3** concerts

What went wrong?

A **long** concert blocked everything else...

$\Rightarrow$ New idea: prefer the **shortest** concerts!

Greedy idea #2: the shortest one



Greedy: picks the short one in the middle $\rightarrow$ **1** concert

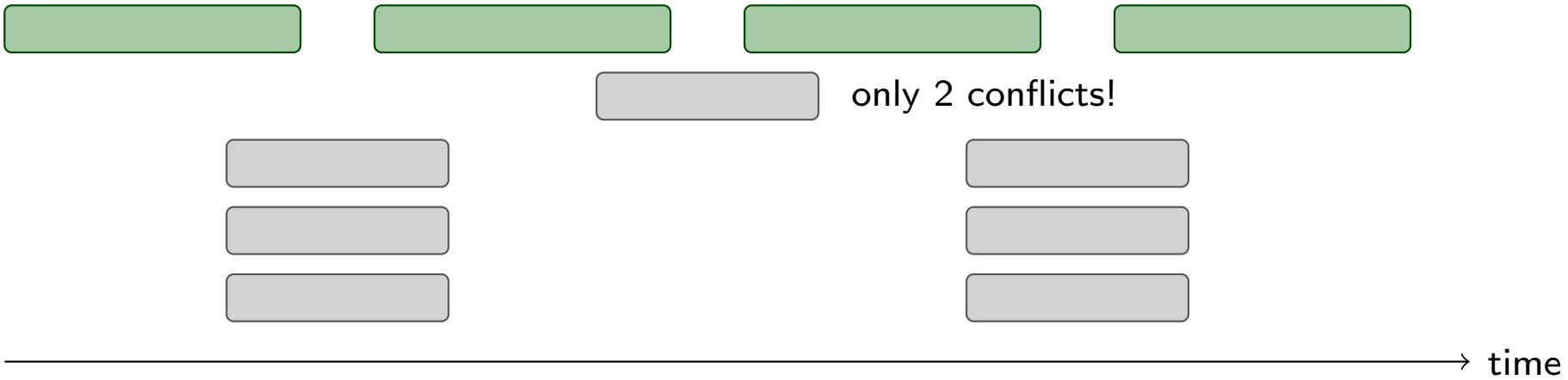
Best: the two long ones $\rightarrow$ **2** concerts

What went wrong?

A short concert can still **clash with many** others...

$\Rightarrow$ New idea: prefer the one with the **fewest conflicts**!

Greedy idea #3: the one with fewest conflicts



Greedy: picks the middle one (2 conflicts)... then only one more fits on each side $\rightarrow$ **3** concerts

Best: the four on the top row $\rightarrow$ **4** concerts (much harder to find!)

What went wrong?

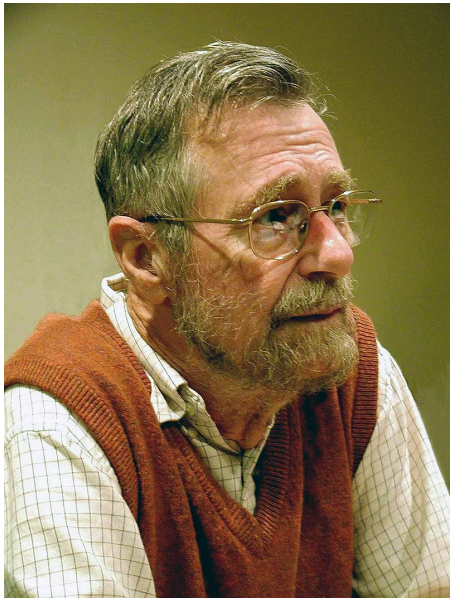
Counting conflicts is not what matters... what matters is **when we are free again**

$\Rightarrow$ New idea: prefer the one that **ends first**!

Greedy idea #4: the one that ends first

- Intuition: "free yourself as soon as possible, to leave the most room for the rest"
- Did anyone find a **counterexample**?
- **No!** This one is actually **correct**
  (it always gives the maximum number of concerts)
- But... **not being able to find a counterexample is not a proof**!
- To be sure we need a **proof** of correctness
  (it exists, and you will see this kind of argument in later courses)

Dijkstra

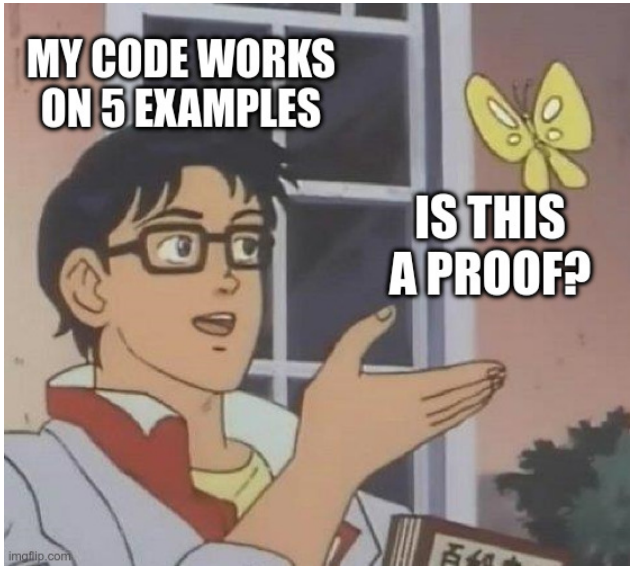


Edsger W. Dijkstra (Wikipedia entry) (Wikiquote)
[1972 Turing Award]

"Program testing can be a very effective way to show the presence of bugs, but is hopelessly inadequate for showing their absence."
(E. W. Dijkstra, *The Humble Programmer*, ACM Turing Lecture, 1972)

Lesson #3: always try to break your idea

- A natural idea that works on the examples **can still be wrong**
- To show that an algorithm is **wrong**: **one** counterexample is enough
- To show that an algorithm is **correct**: no number of examples is enough



- So be your algorithm's **worst enemy**:
  - smallest and largest values; empty and tiny inputs
  - repeated values and ties
  - sorted, reversed, all equal, ...
  - and especially: "what would make **this specific idea** fail?"

Greedy is everywhere

Greedy algorithms

- What we just did has a name: **greedy**
  build the solution step by step, always take what looks best **right now**, and never go back

- Very natural (it is how we often decide things in real life)
- Usually simple and fast

- But **very easy to get wrong**: sometimes it works, sometimes it does not

Whenever you have a greedy idea, your first reaction should be:
"what input would break it?"

Making change



- How to use the **smallest possible number of coins** to make a certain amount?
  (assuming we have an infinite supply of coins)
- **Greedy idea:** use the **largest coin** that is less than or equal to the remaining amount, and repeat
- Example: $3.45€ = 2€ + 1€ + 0.20€ + 0.20€ + 0.05€$ (5 coins)

Making change

- Will this algorithm always produce the minimum number of coins?
- For the euro: yes!
- For a general coin system... no!
- Example: coins $\{1, 5, 6\}$
  - Greedy gives: $10 = 6 + 1 + 1 + 1 + 1$ **(5 coins)**
  - Minimum: $10 = 5 + 5$ **(2 coins)**
- "But that is an invented coin system!"
- UK in the 1960s (before decimal coins): 1, 3, 6, 12, 24 and 30 pence
  - Greedy: $48 = 30 + 12 + 6$ **(3 coins)**
  - Minimum: $48 = 24 + 24$ **(2 coins)**

Making change

Canonical Coin Systems for CHANGE-MAKING Problems

Xuan Cai
Department of Computer Science and Engineering, Shanghai Jiao Tong University
BASICS, MOE-Microsoft Laboratory for Intelligent Computing and Intelligent Systems
Shanghai 200240, China
Email: caixuanfire@sjtu.edu.cn

***Abstract*—The CHANGE-MAKING problem is to represent a given value with the fewest coins under a given coin system. As a variation of the knapsack problem, it is known to be NP-hard. Nevertheless, in most real money systems, the greedy algorithm yields optimal solutions. In this paper, we study what type of coin systems that guarantee the optimality of the greedy algorithm. We provide new proofs for a sufficient and necessary condition for the so-called *canonical* coin systems with 4 or 5 types of coins, and a sufficient condition for non-canonical coin systems, respectively. Moreover, we propose an $O(m^2)$ algorithm that decides whether a tight coin system is canonical.**

The CHANGE-MAKING problem is NP-hard [6], [7] by a polynomial reduction from the knapsack problem. There are a large number of pseudo-polynomial exact algorithms [8] solving this problem, including the one using dynamic programming [11]. However, the greedy algorithm, as a simpler approach, can produce the optimal solutions for many practical instances, especially canonical coin systems.

Definition 1: A coin system $ is canonical if $|\mathrm{GRD}_{\$}(x)| = |\mathrm{OPT}_{\$}(x)|$ for all x.

- Coin systems for which greedy **always** gives the minimum are called **canonical**
- **Question:** given a coin system, is it canonical?
  (should we trust greedy for these coins, or not?)
- Papers like this one give **conditions** and **algorithms** to answer that question
- Knowing **when** greedy works is an actual studied problem in algorithms!

Cai, X. (2009). Canonical coin systems for change-making problems. In *2009 Ninth International Conference on Hybrid Intelligent Systems* (pp. 499–504). IEEE. https://doi.org/10.1109/HIS.2009.103

It happens to everyone

Microsoft Zune (December 31, 2008)

- On the last day of 2008, **all** Zune 30 music players froze at the same time
- The culprit: code converting "days since January 1, 1980" into the current year
  (here translated to Python)

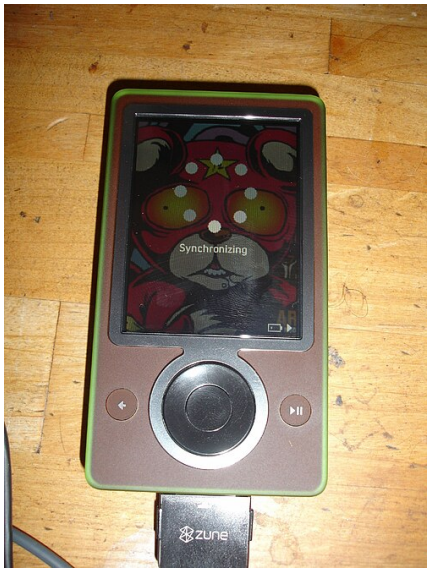


Photo: Travis Hornung, CC BY 2.0

```
year = 1980
while days > 365:
    if is_leap(year):
        if days > 366:
            days -= 366
            year += 1
    else:
        days -= 365
        year += 1
```

- 2008 was a leap year: on its **last day**, the loop got to days = 366...
- ...and the loop never ends! (nothing changes when days is 366)
- Official fix: **wait until tomorrow** (really)

Scientists too

- Even **published** algorithms, read and checked by experts, can be wrong!

CORRECTIONS TO LEE'S VISIBILITY POLYGON ALGORITHM

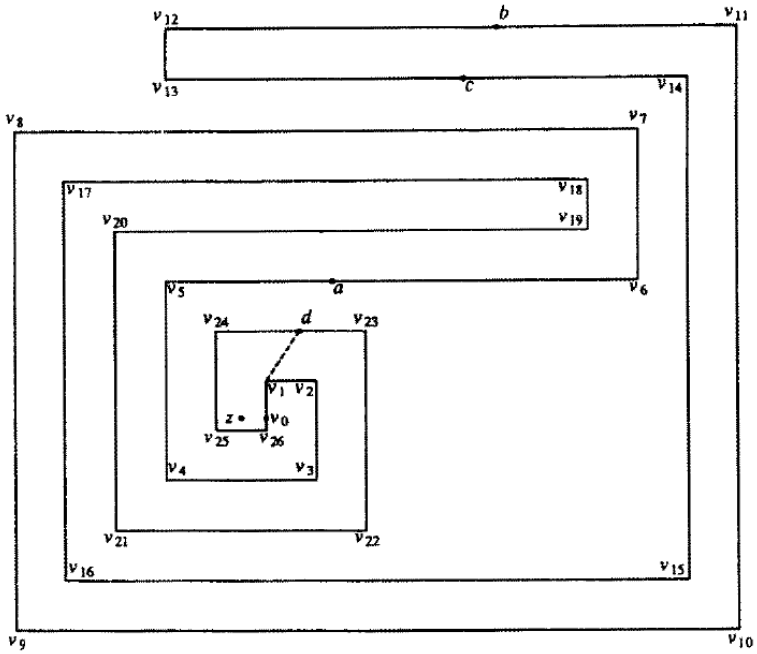
B. JOE and R. B. SIMPSON

Dept. of Computing Science, University of Alberta, Edmonton, Alberta, Canada T6G 2H1

Dept. of Computer Science, University of Waterloo, Waterloo, Ontario, Canada N2L 3G1

Abstract.

We present a modification and extension of the (linear time) visibility polygon algorithm of Lee. The algorithm computes the visibility polygon of a simple polygon from a viewpoint that is either interior to the polygon, or in its blocked exterior (the cases of viewpoints on the boundary or in the free exterior being simple extensions of the interior case). We show by example that the original algorithm by Lee, and a more complex algorithm by El Gindy and Avis, can fail for polygons that wind sufficiently. We present a second version of the algorithm, which does not extend to the blocked exterior case.

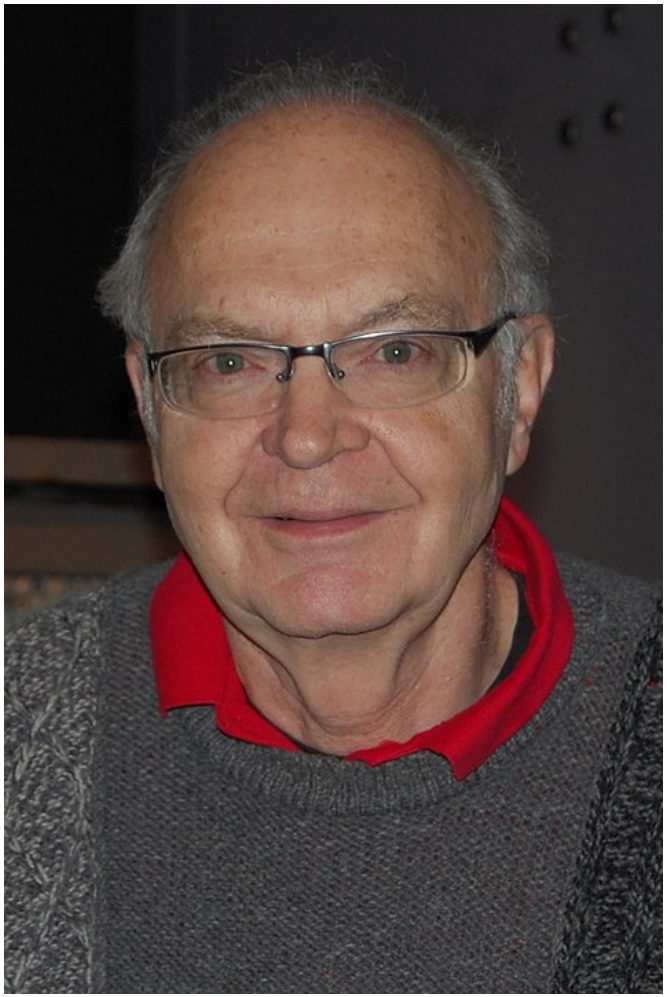


(one of the counterexamples in the paper)

- **Problem:** given a polygon and a point inside it, what part of the polygon can you **see** from that point?
- Two published algorithms (Lee, 1983; El Gindy & Avis, 1981) **fail** for polygons that "wind" enough

Joe, B., & Simpson, R. B. (1987). Corrections to Lee's visibility polygon algorithm. *BIT Numerical Mathematics*, *27*(4), 458–473. https://doi.org/10.1007/BF01937271

Even when you prove it...



Donald Knuth (Wikipedia entry) (Wikiquote)
"father" of the analysis of algorithms
[1974 Turing Award]

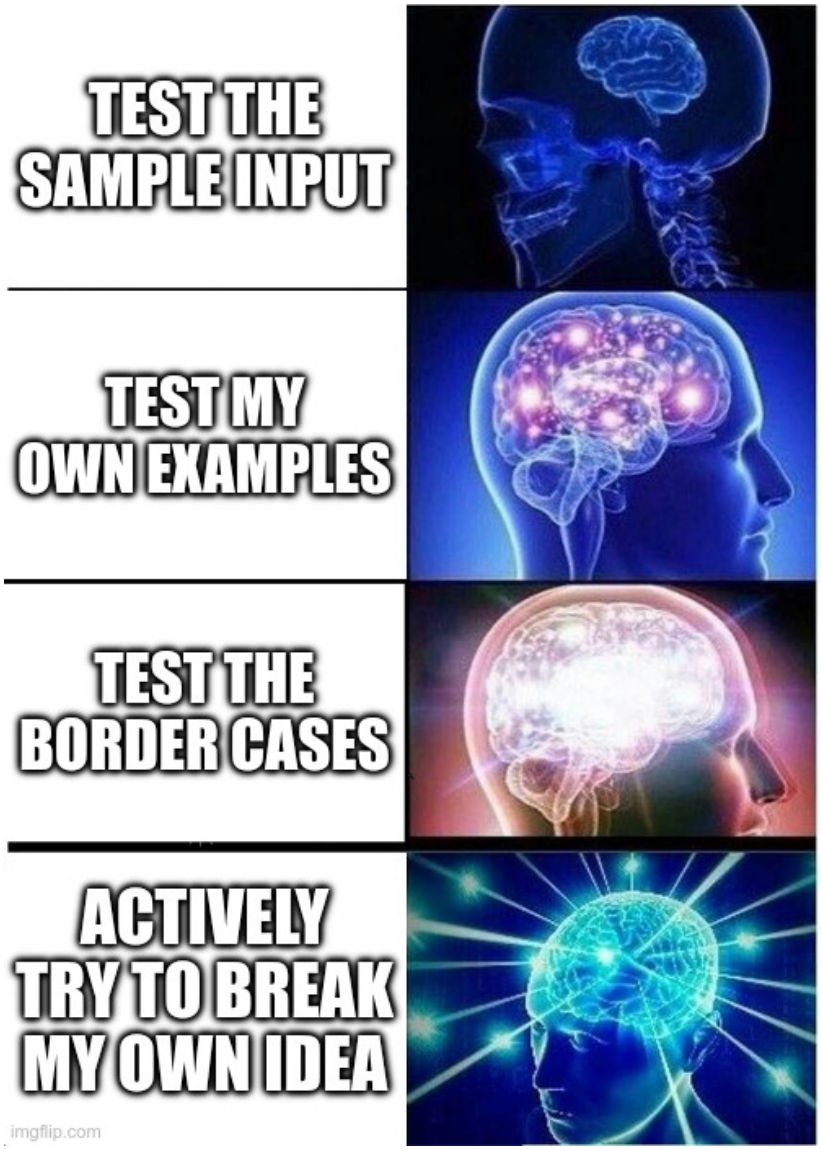
"Beware of bugs in the above code; I have only proved it correct, not tried it."
(D. E. Knuth, note to P. van Emde Boas, 1977)

(proofs and tests complement each other: the proof is about the *idea*, the test is about the *code*)

Take-home messages about correctness

- Three ways of being wrong: **inputs**, **code**, **idea**
- Know your instances, **and the right answer for each**: read the problem statement, think about border cases, test your own program
- Try to **break** your idea, and **attack again** after every fix
- One counterexample is enough to show it is wrong; no number of tests shows it is right
  (not even an Accepted on Mooshak!)

Take-home messages about correctness

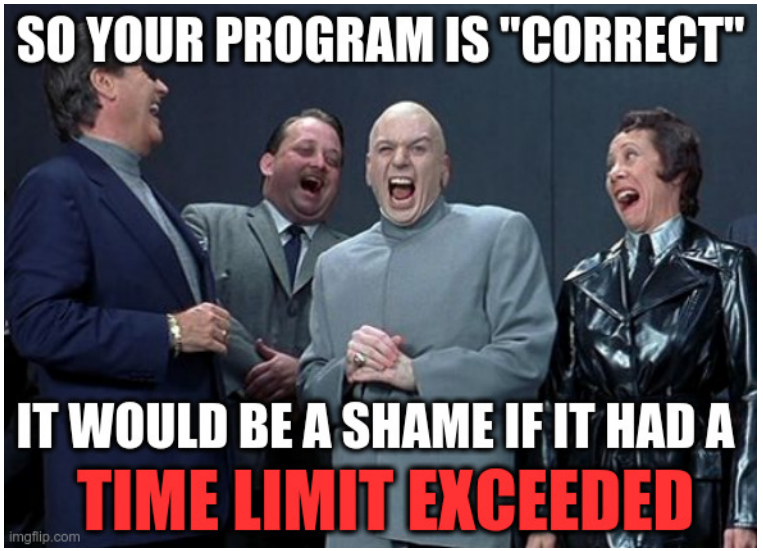


Coming next

Is it fast enough... for every input?

- Today: is it **right**?
  Friday: is it **fast enough**?

- You have already met efficiency today... without noticing:
  - **Sensor data:** removing in place *vs* building a new list: which is faster?
  - **Silver medal:** we sorted the **whole** list just to find the second largest... is that wasteful?
  - **Coins:** greedy can fail... what if we simply **try every combination**?
    Always correct! But...



See you on Friday! (efficiency)