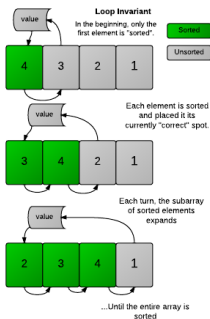


# Correctness and Loop Invariants

Pedro Ribeiro

DCC/FCUP

2018/2019



# On Algorithms


What are algorithms? A set of **instructions** to solve a **problem**.

- The problem is the **motivation** for the algorithm
- The instructions need to be **executable**
- Typically, there are **different algorithms** for the same problem [how to choose?]
- **Representation**: description of the instructions that is understandable for the intended audience

*My favourite dish* Pasta with bacon and tomato sauce

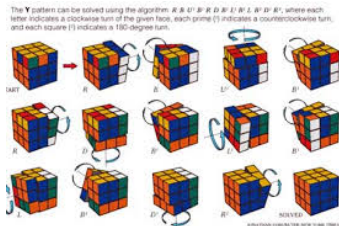
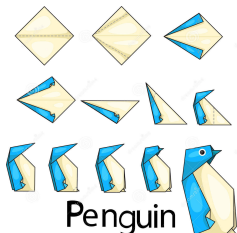
**Ingredients**

- 1 red onion
- 2 red peppers
- 120 g bacon
- 1 can (450 g) tomatoes
- 1 cup water
- olive oil
- garlic
- oregano
- 50 g pasta per person



**Method**

- 1 Cut the onion, red peppers and bacon into small pieces.
- 2 Heat some olive oil in a pan and fry the onion, red peppers and bacon.
- 3 Add **oregano, garlic, tomatoes and water** and cook for 20 minutes.
- 4 Cook the pasta in a big pot of boiling water.
- 5 Serve the pasta with the sauce, and enjoy!



# On Algorithms

## "Computer" Science version

- An algorithm is a **method** for solving a (computational) problem
- Algorithms are the **ideas** behind the programs and are independent from the programming language, the machine, ...
- A **problem** is characterized by the description of its **input** and **output**

A classical example:

### Sorting Problem

**Input:** a sequence of  $\langle a_1, a_2, \dots, a_n \rangle$  of  $n$  numbers

**Output:** a permutation of the numbers  $\langle a'_1, a'_2, \dots, a'_n \rangle$  such that  $a'_1 \leq a'_2 \leq \dots \leq a'_n$

### Example instance for the sorting problem

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

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

# On Algorithms

What do we aim for?

- What **properties** do we want on an algorithm?

## Correction

It has to solve correctly **all instances** of the problem

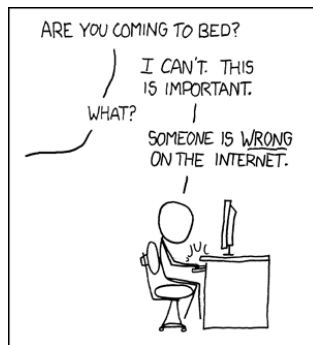
## Efficiency

The performance (**time** and **memory**) has to be adequate

- This course is about **designing** correct and efficient algorithms and how to **prove** they meet the specifications

# About correction

- In this lecture we will (mostly) worry about **correction**
  - ▶ Given an algorithm, it is not often obvious or trivial to know if it is **correct**, and even less so to **prove** this.
  - ▶ By learning how to reason about correctness, we also gain **insight** into what really makes an algorithm work



# Loops

- We will tackle one of the most fundamental (and most used) algorithmic patterns: a **loop** (e.g. `for` or `while` instructions)

## Example loop: summing integers from 1 to $n$

```
sum = 0
i = 1
while (i ≤ n) {
    sum = sum + i
    i = i + 1
}
```

- We will talk about how to prove that a **loop** is correct
- We will show how this is also useful for **designing** new algorithms

# Loop Invariants

## Definition of Loop Invariant

A **condition** that is necessarily true immediately before (and immediately after) each iteration of a loop

Note that this says nothing about its truth or falsity part way through an iteration.

*Instructions are for computers, invariants are for humans*

- The loop program statements are "**operational**", they are "**how to do**" instructions
- Invariants are "**assertional**", capturing "**what it means**" descriptions

# Anatomy of a loop

Consider a simple loop: **while (B) { S }**

- **Q**: precondition (assumptions at the beginning)
- **B**: the stop condition (defining when the loop end)
- **S**: the body of the loop (a set of statements)
- **R**: postcondition (what we want to be true at the end)

**Example loop: summing integers from 1 to  $n$**

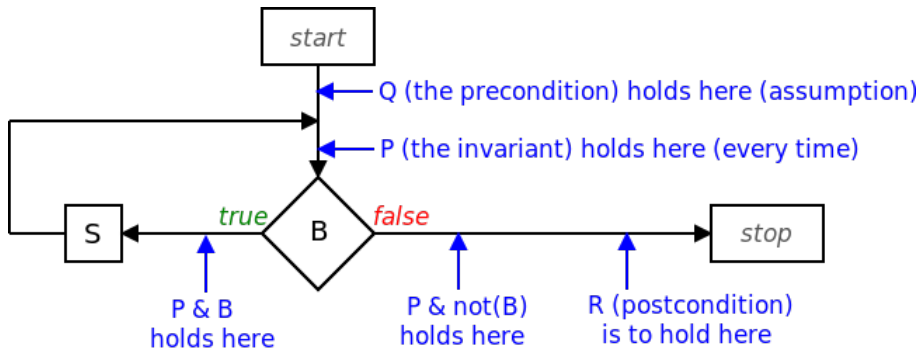
```
sum = 0
i = 1
while (i ≤ n) {
    sum = sum + i
    i = i + 1
}
```

- **Q**:  $sum = 0$  and  $i = 1$
- **B**:  $i \leq N$
- **S**:  $sum = sum + i$  followed by  $i = i + 1$
- **R**:  $sum = \sum_{i=1}^n i$



# The invariant?

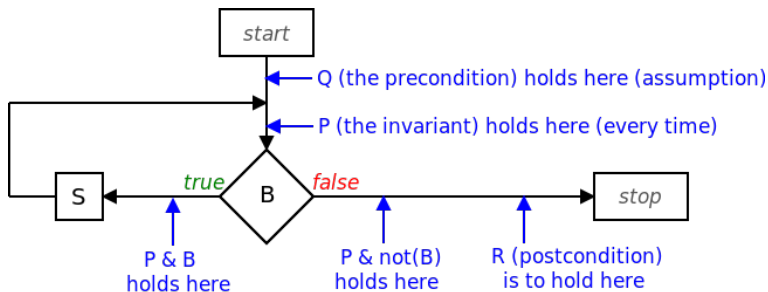
- **P**: an invariant (condition that holds at the start of each iteration)



- To be **useful**, the invariant  $P$  that we seek should be such that:  
 $P \wedge \text{not}(B) \rightarrow R$

- ▶ For the example sum loop, it could be:  $sum = \sum_{i=1}^{i-1} i$

# How to show that an invariant is really one?



- First, show that  $Q \rightarrow P$   
(truth precondition  $Q$  guarantees truth of invariant  $P$ )
  - ▶ For the example sum loop:  $\text{sum}=0$  which is  $= \sum_{i=1}^0 i$
- If  $P \wedge B$ , then after executing  $S$ , then  $P$  holds after executing  $S$   
(the statements  $S$  of the loop guarantee that  $P$  is respected)
  - ▶ For the example sum loop:  $\sum_{i=1}^{i-1} + i = \sum_{i=1}^i$

# How to show that an invariant is really one?

## Initialization

The invariant is true prior to the first iteration of the loop

## Maintenance

If it is true before an iteration of the loop, it remains true before the next iteration

## Termination

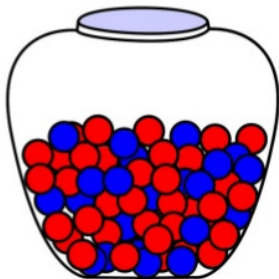
When the loop terminates, the invariant gives us a useful property that helps show that the algorithm is correct

# How to show that the loop terminates?

- We need to show that each iteration **makes progress towards termination** in some way
- This is typically done by choosing an **integer function** that keeps getting closer (i.e., decreasing or increasing) towards the stop condition
  - ▶ For the example sum loop: we could simply use the value of  $i$ , which keeps getting closer to  $n$

# Motivation: a small puzzle

Suppose you have a jar of one or more marbles, each of which is either **RED** or **BLUE** in color.



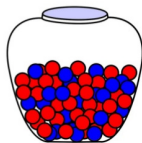
# Red and Blue Marbles in a Jar

Suppose you have a jar of one or more marbles, each of which is either **RED** or **BLUE** in color. You also have an unlimited supply of **RED** marbles off to the side. You then execute the following "procedure":

## Red and Blue Marbles in a Jar

```
while (# of marbles in the jar > 1) {
  choose (any) two marbles from the jar;
  if (the two marbles are of the same color) {
    toss them aside;
    place a RED marble into the jar;
  } else { // one marble of each color was chosen
    toss the chosen RED marble aside;
    place the chosen BLUE marble back into the jar;
  }
}
```

# Red and Blue Marbles in a Jar



- Does it **terminate**?
- Let  $f(n)$  be the number of marbles in the jar
- After each iteration,  $f(n)$  decreases exactly by one
- When  $f(n) \leq 1$ , the loop stops

# Red and Blue Marbles in a Jar

- Suppose we know the initial contents of the jar (number of marbles of each color)
- Can we **predict** which will be the last marble left in the jar?
- More formally, we need a function  $f : \mathbb{N} \times \mathbb{N} \rightarrow \{RED, BLUE\}$
- It turns that this function exists! The key to identifying it is to first identify an **invariant** of the loop having to do with the **number of BLUE** marbles in the jar
- Consider the effect of one iteration:
  - ▶ If both marbles chosen are the same, the number of blue marbles either stays the same or decreases by two
  - ▶ If the marbles are different, the number of blue marbles stays the same
- An iteration does not affect the **parity** of the number of blues!
  - ▶ If it was odd, it stays odd
  - ▶ If it was even, it stays even



# Red and Blue Marbles in a Jar

- $A$ : initial number of blue marbles
- $B$ : final number of blue marbles

## Invariant

$B$  is odd if and only if  $A$  is odd

This is the same saying that both  $A$  and  $B$  are odd, or both are even

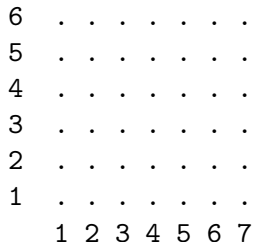
- Because at the end we are left with one marble either  $B = 0$  or  $B = 1$
- So, if  $A$  is even, at the end  $B = 0$  (the remaining marble is **RED**)
- If  $A$  is odd, then at the end  $B = 1$  (the remaining marble is **BLUE**)

Thus  $F(A, -) = \{RED \text{ if } A \text{ is even, BLUE otherwise}\}$

*Interestingly, the color of the last remaining marble does not depend at all upon the number of RED marbles initially in the jar.*

# Motivation: a drawing game

There are two players, call them Red and Blue. The game is played on a rectangular grid of points, such as the one illustrated below.



- Red and Blue take alternating turns, each time drawing an horizontal or vertical line segment of their color connecting two unconnected points
- Red's goal is to form a closed curve
- Blue wants to prevent this from happening
- The game ends when red wins, or when no more segments can be drawn

# Motivation: a drawing game

The game can be seen as a loop:

## Red and Blue Marbles in a Jar

```
while (more line segments can be drawn) {  
    Red draws line segment;  
    Blue draws line segment;  
}
```

- Does either Red or Blue have a **"winning strategy"**?

# Motivation: a drawing game

## A winning strategy for the blue player

- Suppose red connects  $(i, j)$  with  $(i, j + 1)$  (horizontal line)
  - ▶ Red responds by connecting  $(i - 1, j + 1)$  with  $(i, j + 1)$  (if  $i = 1$ , then draw anywhere)
- Suppose red connects  $(i, j)$  with  $(i + 1, j)$  (vertical line)
  - ▶ Red responds by connecting  $(i + 1, j - 1)$  with  $(i + 1, j)$  (if  $j = 1$ , then draw anywhere)
- Informally, Blue responds to Red by making sure that the line segment just drawn by Red can never occur as one of the two line segments forming an "**upper right corner**" of a closed curve of red segments.
- Note that any closed curve of red line segments must include at least one such corner. Thus, if Blue adheres to this strategy, Red can never form a closed curve!

# Motivation: a drawing game

Assuming that Blue follows this strategy, the following statement is true after each step in the playing of the game:

## Invariant

There does not exist on the grid a pair of red line segments that form an upper right corner.

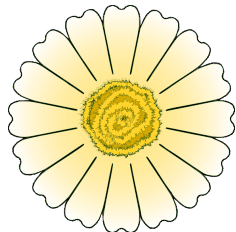
Of course, there is nothing special about upper right corners. Blue could have just as easily chosen to prevent Red from forming any of the other three kinds of corners instead.

Significantly, when the game ends (i.e., the loop terminates), the invariant will hold, and Red will not have formed an upper right corner. Which means that Red must not have won, so Blue must have won.

# A puzzle for you to solve

Imagine a flower having 16 petals. Two players take alternating moves. A move involves removing either one petal or two adjacent petals from the flower, at the player's choice. The winner is the one removing the last petal.

**Question:** Does either player have a **winning strategy**?



And if we have  $n$  petals?

# Back to computer programs

*We will now interactively create some loop based code, showing is design, what are the invariants, and how can we prove it is correct - I'll add that material here later*