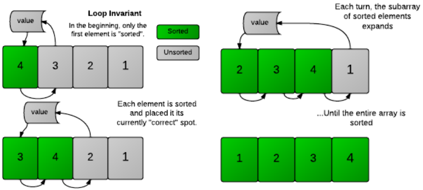
Correctness and Loop Invariants

L.EIC

Algoritmos e Estruturas de Dados

2024/2025

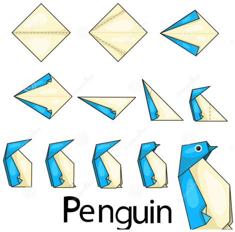


P Ribeiro, AP Tomás

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




```python
# Summing all integers from 1 to n
def my_sum(n):
    return n*(n+1)//2
```

```c
// Summing all integers from 1 to n
int my_sum(int n) {
    return n*(n+1)/2;
}
```

On Algorithms

"Computer Science" version

- An algorithm is a **method** for solving a (computational) problem
- A **problem** is characterized by the description of its **input** and **output**

A classical example:

Sorting Problem

Input: a sequence of $\langle a_1, a_2, \ldots, a_n \rangle$ 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 instance for the sorting problem

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

On Algorithms

What properties do we want on an algorithm?

Algorithm

Well-defined computational procedure that takes some value, or set of values, as *input* and produces some value, or set of values, as *output*. The number of steps must be finite.

Correctness

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

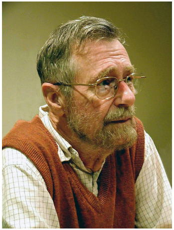
Instance: example of a concrete and valid input.

Efficiency

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

This course is about **being able to choose and design** correct and efficient algorithms.

Dijkstra



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

"How do we convince people that in programming **simplicity and clarity** - in short: what mathematicians call **"elegance"** — are not a dispensable luxury, but a crucial matter that decides between success and failure?"

"Program testing can be a very effective way to show the presence of bugs, but it is hopelessly inadequate for showing their absence."

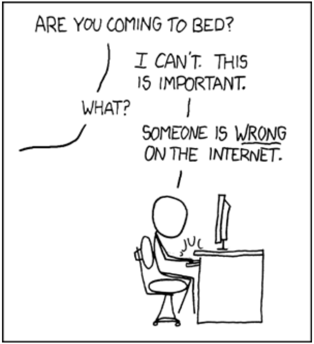
Algorithm Analysis

	theoretical analysis	**experimental analysis**
correctness	proof or correctness argumentation	predefined or randomized tests
efficiency (time and space)	complexity and asymptotic analysis	performance tests

"Testing shows the presence, not the absence of bugs" - **Edsger Dijkstra**
(a more succinct form of the previous quote)

About correctness

- In the this lecture we will (mostly) worry about **correctness**
  - 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)

```
// Summing all integers from 1 to n (using a loop, not n*(n+1)/2)
int sum = 0, i;
for (i=1; i<=n; i++)
   sum += i;
```

```
// Equivalently with a while
int sum = 0;
int i = 1;
while (i<=n) {
   sum +=i;
   i++;
}
```

- 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.

- 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** $\}$

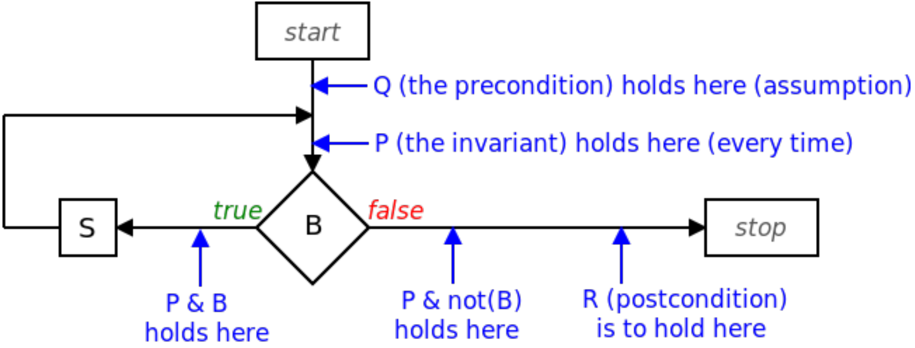
- **Q**: precondition (assumptions at the beginning)
- **B**: the stop condition (defining when the loop ends)
- **S**: the body of the loop (a set of statements)
- **R**: postcondition (what we want to be true at the end)

```
// Summing all integers from 1 to n
int sum = 0;
int i = 1;
while (i<=n) {
   sum +=i;
   i++;
}
```

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

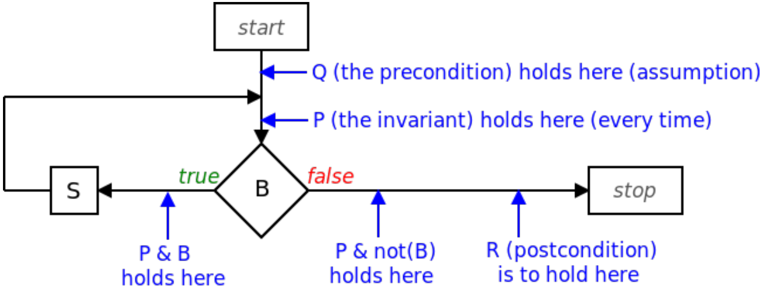
The invariant?

- **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: $P \wedge not(B) \rightarrow R$
  - For the example sum loop, it could be: $sum = \sum_{k=1}^{i-1} k$

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: sum=0 which is $= \sum_{k=1}^{0} k$
- If $P \wedge B$, then P holds after executing S
  (the statements S of the loop guarantee that P is respected)
  - For the example sum loop: $\left(\sum_{k=1}^{i-1} k\right) + i = \sum_{k=1}^{i} k$

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
  - The loop ends when $i = n + 1$. Therefore, using the invariant, we conclude that:

$$\texttt{sum} = \sum_{k=1}^{i-1} k = \sum_{k=1}^{(n+1)-1} k = \sum_{k=1}^{n} k$$

Some additional comments

Correctness of algorithms versus programs

```
// Summing all integers from 1 to n
int my_sum(int n) {
    return n/2*(n+1);          // Wrong (7/2*8 = 24 but 7*8/2 = 28)
}
```

```
int my_sum(int n) {
    return n*(n+1)/2;       // Correct:  n*(n+1)/2 = (n*(n+1))/2
}
```

```
int my_sum(int n) {    // Correct but not efficient (too many operations)
    int sum = 0;
    for(int i = 1; i <= n; i++)  sum += i;
    return sum;
}
```

In Maths: $\frac{n}{2}(n+1) = \frac{n(n+1)}{2} = n\frac{n+1}{2} = \sum_{i=1}^{n} i$

In C++: `n*(n+1)/2` is not equivalent to `n/2*(n+1)`. Recall: $7/\ 2 = 3$

Steps to a proof using an invariant

• 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

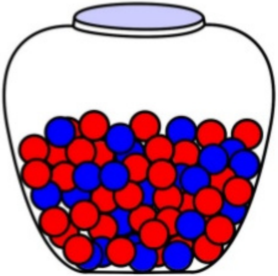
We also need to show that the loop terminates:

• Progress

Each iteration gets us closer to the end until eventually we finish

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

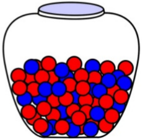
```
// selects two balls at random; Returns (0,0), (0,1), (1,1), or (1,0)
pair<int,int> select(int red,int blue);      // Definition not shown

// playing the game
pair<int,int> play(int red, int blue) {
    assert(red > 0 && blue > 0); // Ensure both positive

    while (red + blue > 1) {
        pair<int,int> selected = select(red, blue);
        int s1 = selected.first;
        int s2 = selected.second;
        if (s1 + s2 == 1  ||  s1 + s2 == 0) {  // Distinct or RED
            red -= 1;
        } else {  // Both BLUE
            blue -= 2;
            red += 1;
        }
    }
    return make_pair(red,blue); // Return the remaining balls
}
```

Does play terminate? Can we predict the return value?

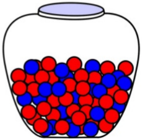
Red and Blue Marbles in a Jar



- 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



Let's state it a bit more formally...

- Let $f(n)$ be the number of marbles in the jar when we start iteration n. In the program, $f(n)$ = `red+blue`
- After each iteration, $f(n)$ decreases exactly by one, i.e, $f(n+1) = f(n) - 1$. (Check the update of `red` and `blue`)
- When $f(n) \leq 1$, the loop stops
- If `red` ≥ 1 and `blue` ≥ 1 at start, then $f(n) = 1$ when 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: (current) number of blue marbles at the start of an iteration

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 $F(\_, A) = \{RED \textbf{ if A is even, } BLUE \textbf{ otherwise}\}$

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

Back to computer programs

In order to prove the correctness of a loop using invariants, we must first **find a suitable loop invariant** condition and then show the following three things:

- **Initialization:** It 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.

We also need to show that the loop terminates:

- **Progress:** Each iteration gets us closer to the end until eventually we finish

Useful loop invariants?

```
int func1(int n) {
   int i=1, s;
   while (i < n/2) {
      s += 2;  n = n/2;
   }
   return s;
}
```

```
int func2(int n) {
   int s = 0;
   while (1 < n/2) {
      s += 2;  n = n/2;
   }
   return s;
}
```

- $i = 1$ is a loop invariant in `func1`. How helpful is it?
- A more interesting loop invariant that is true for both: *for all $k \geq 1$, if we are testing the loop condition for the kth time then $s = s_0 + 2(k-1)$ and $n = n_0/2^{k-1}$, being s_0 and n_0 the values of s and n before the loop.*
- Although we cannot say anything about the value of s_0 for `func1`, that invariant can be proved.

For `func2`, we have $s_0 = 0$ and $\texttt{func2(n)} = \begin{cases} 2\lfloor \log_2(\texttt{n}) \rfloor - 2 & \text{if } \texttt{n} > 3 \\ 0 & \text{otherwise} \end{cases}$

A simple example - checking if a number is prime

Let's try to prove the following program is correct:

```
// for simplicity, assume n is an integer > 1
bool isPrime(int n) {
   for (int i = 2; i*i <= n; i++)     // why i*i <= n?
      if (n % i == 0) return false;
   return true;
}
```

- **Invariant**: (start of an iteration)
  there is no divisor d of n such that $1 < d < i$
- **Initialization:** at the beginning, $i = 2$, and therefore trivially there is no d such that $d > 1$ and $d < 2$. (so, there is no divisor d such that $1 < d < 2$).
- **Maintenance:** start of the iteration i: the invariant is true (no divisors $1 < d < i$); if loop continues, i does not divide n; therefore at the start of iteration $i + 1$ the invariant is still true (no divisors $1 < d < i + 1$)
- **Termination:** Either the loop terminated early (and we found a divisor $d \leq \sqrt{n}$) or we know that there are no such divisors and therefore the number must be prime
- **Progress:** i is increased until it surpasses $\sqrt{n}$

Checking if a number is prime

Why it is correct to stop when $i \times i > n$

Theorem (that supports our argument): For all $n, i \in \mathbb{Z}^+$, if i divides n then n/i divides n. Thus, $n \in \mathbb{Z}^+$ is a prime number iff $n \neq 1$ and there is no integer i such that $2 \leq i \leq \sqrt{n}$ and i divides n.

Pairing positive divisors for n=30 and for n=100

```
1                   30        1                   100
  2              15             2              50
    3         10                  4         25
      5   6                         5    20
                                      10
```

- Thus, if $i > n/i$, or equivalently if $i^2 > n \Leftrightarrow i > \sqrt{n}$, then n/i has been checked already (therefore, i has been checked implicitly).

Examples of proofs

Problem 1: Find the maximum of an array

Write a function $\text{PosMax}(v, n)$ that finds the position of the first occurrence of the maximum value of $v[1], v[2], \ldots, v[n]$ in v, for $n \geq 1$.

Consider 3 cases:

- **(Case a)** v is not sorted.
- **(Case b)** We know that $v[1] < v[2] < \ldots < v[n]$.
- **(Case c)** We know that $v[1] \leq v[2] \leq \ldots \leq v[n]$.

Proof of correctness - Example 1b)

Problem 1b):

To find the position of the first occurrence of the maximum value of v, assuming that $v[1] < v[2] < \ldots < v[n]$.

$\textsc{PosMaxSorted}(v, n)$
    return n;

Correctness: Under the assumption that $v[1] < v[2] < \ldots < v[n]$, it is trivial to conclude that n is the correct answer.

Time complexity: The running time does not depend on the input size. Later, we will describe it as $O(1)$.

Proof of correctness - Example 1a)

Problem 1a):

To find the position of the first occurrence of the maximum value of v, which can be in any order.

```
PosMax(v, n)
1   imax ← 1;
2   i ← 2;
3   while i ≤ n do
4.      if v[i] > v[imax] then
5.          imax ← i;
6.      i ← i + 1;
7.  return imax;
```

Loop Invariant: At **line 3**, when we are testing the condition for the k-th time, with $k \geq 1$, we have $2 \leq i = k + 1 \leq n + 1$, the index of the first occurrence $\max(v[1], v[2] \dots v[k])$ is *imax*, we have not analysed $v[k+1], \dots, v[n]$. The value of n does not change in the loop.

Termination: The loop ends with $i = n + 1$ and *imax* is the index of the first occurrence of $\max(v[1], \dots, v[n])$. So, *imax* has the correct value at **line 7**.

Proof of correctness - Example 1a)

How can we **prove that the invariant is valid**?

Loop Invariant: At **line 3**, when we are testing the condition for the k-th time, with $k \geq 1$, we have $2 \leq i = k + 1 \leq n + 1$, the index of the first occurrence $\max(v[1], v[2] \ldots v[k])$ is *imax*, we have not analysed $v[k+1], \ldots, v[n]$. The value of n does not change in the loop.

Sketch of the Proof by induction on k

If we show conditions (1) and (2) then, by the **induction principle**, it follows that the property is true for all $k \geq 1$.

1. Initialization or *Base case*:
   The property (invariant) holds for $k = 1$.

2. Maintenance or *Induction step* or Inheritance
   For all $k \geq 1$, **if** the property holds at iteration k **then** it holds at iteration $k + 1$.

Proof of correctness - Example 1a)

Loop Invariant: At **line 3**, when we are testing the condition for the k-th time, with $k \geq 1$, we have $2 \leq i = k + 1 \leq n + 1$, the index of the first occurrence $\max(v[1], v[2] \ldots v[k])$ is *imax*, we have not analysed $v[k+1], \ldots, v[n]$. The value of n does not change in the loop.

Proof by induction on k

1. Initialization or *Base case*:

   The property (invariant) holds for $k = 1$.

   Indeed, the values of *imax* and i are 1 and 2, when we start the loop, and $2 \leq i = k + 1 \leq n + 1$, if we assume $n \geq 1$.

   So, the value of *imax* is the index of $\max(v[1])$, which is 1. It is true that we have not analysed $v[2], \ldots, v[n]$ yet. $\square$

Proof of correctness - Example 1a)

Loop Invariant: At **line 3**, when we are testing the condition for the k-th time, with $k \geq 1$, we have $2 \leq i = k + 1 \leq n + 1$, the index of the first occurrence $\max(v[1], v[2] \ldots v[k])$ is *imax*, we have not analysed $v[k+1], \ldots, v[n]$, and n is constant along the loop.

Proof by induction on k

1. Maintenance or *Induction step* or Inheritance

   For all $k \geq 1$, **if** the property holds at iteration k **then** it holds at iteration $k + 1$.

   By the induction hypothesis, the property holds at iteration k. So, if we are testing the condition for the $(k+1)$th time then, when the iteration k started, we had $i = k + 1 \leq n$, and *imax* contained the index of the first occurrence of $\max(v[1], \ldots, v[i-1])$ and $v[i], \ldots, v[n]$ had not been checked yet.

   In iteration k, we changed *imax* to i, if $v[i] > v[imax]$, which is correct because $v[i] > v[imax] = \max(v[1], \ldots, v[i-1])$. After this update, *imax* contains the index of the first occurrence of $\max(v[1], \ldots, v[i-1], v[i])$. If $v[i] \leq v[imax]$, we keep *imax* unchanged, which is correct as, by the hypothesis, *imax* contains the index of the first occurrence of $\max(v[1], \ldots, v[i-1])$, which is the same for $\max(v[1], \ldots, v[i-1], v[i])$.

   In line 6, we increase i by 1. Thus, when we test the condition on line 3 for the $(k+1)th$ time, we have

   $2 \leq i = (k+1) + 1 \leq n + 1$ and the invariant holds at iteration $k + 1$. □

Proof of correctness – Case 1c)

Problem 1c):

To find the position of the first occurrence of the maximum value of v, assuming that $v[1] \leq v[2] \leq \ldots \leq v[n]$.

How can we **prove** that the following function is correct?

$\textsc{PosMaxSorted}(v, n)$

$i \leftarrow n - 1;$
while $i \geq 1 \wedge v[i] = v[i+1]$ do
    $i \leftarrow i - 1;$
return $i + 1;$

Can we state a useful loop invariant?

By "useful" we mean that it helps us show that the function computes the correct answer...

Proof of correctness – Case 1c)

Problem 1c):

To find the position of the first occurrence of the maximum value of v, assuming that $v[1] \leq v[2] \leq \ldots \leq v[n]$.

How can we **prove** that the following function is correct?

```
PosMaxSorted(v, n)
1.    i ← n − 1;
2.    while i ≥ 1 ∧ v[i] = v[i + 1] do
3.        i ← i − 1;
4.    return i + 1;
```

Loop Invariant: When we are testing the condition in **line 2** for the k-th time, for $k \geq 1$, the value of i is $n - k$, we have not analysed $v[1], \ldots, v[i]$ yet and we know that $v[i+1] = v[i+2] = \ldots = v[n]$ and $i = n - k \geq 0$. The sequence in $v[]$ has not changed.

Therefore, the loop <u>terminates</u> and $i + 1$ at line 4 is the correct answer. **Why?**

Note that, when the loop stops, **either** $i = 0$ **or** $i \geq 1 \wedge v[i] \neq v[i+1]$

Proof of correctness – Case 1c) (cont)

```
PosMaxSorted(v, n)
1.  i ← n − 1;
2.  while i ≥ 1 ∧ v[i] = v[i + 1] do
3.      i ← i − 1;
4.  return i + 1;
```

- Indeed, the loop stops when **either** $i = 0$ **or** $i \geq 1 \wedge v[i] < v[i+1]$.
  (because $v[i] \leq v[i+1]$ for every instance in case 1c))
- The invariant says that $v[i+1] = v[i+2] = \ldots = v[n]$.
- In line 4, **either** $i = 0$ **or** $i \geq 1 \wedge v[i] < v[i+1]$
  The function returns the correct value, because, from the invariant, we conclude that:
  - if $i = 0$ then $v[1] = v[2] = \ldots = v[n]$. The index of the first occurrence of $\max(v[1], v[2], \ldots, v[n])$ is 1, which is $i + 1$.
  - if $i \geq 1$ then $v[i] < v[i+1] = v[i+2] = \ldots v[n]$. So, the index is $i + 1$.

Final Remarks

- **Invariants** capture the **"semanting meaning"** of loops, the logic and intuition behind them
- Thinking about invariants and their properties they will **help you reason** about a correct solution
- Along the course sometimes we will refer to invariants to help you **understand how** an algorithm works and **why** it is correct

- Correctness is often not trivial to prove are there are many other methodologies, but thinking about it and understanding a proof will give you crucial **insight**

  "If you want more effective programmers, you will discover that they should not waste their time debugging, they should not introduce the bugs to start with." - **Edsger Dijkstra**

- Presentation based on **CLRS** textbook. Not a formal representation of the semantics, required for automated reasoning: e.g, Hoare logic, other notions, like partial and total correctness, loop invariants and variants, . . .