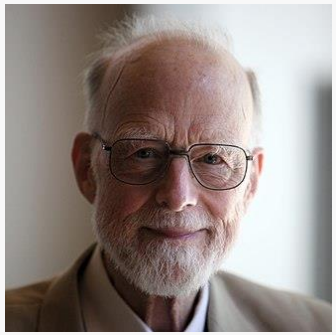
Algorithm correctness analysis

Algoritmos e Estruturas de Dados,
L.EIC, 2023/2024

P Diniz, AP Rocha,
A Costa, B Leite, F Ramos, J Pires, PH Diniz, V Silva

Algorithm design

"There are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies. The first method is far more difficult."



C. A. R. Hoare (Tony Hoare)
(1980 Turing Award)
(inventor of quick sort)

Algorithm analysis

- **Correctness**: an algorithm is correct if behaves as specified
- **Efficiency:** performance of the algorithm (time and space)

	theoretical analysis	*experimental analysis*
correctness	proof or correctness argumentation	pre-defined or random tests*
efficiency (time and space)	complexity	performance tests

* "testing shows the presence, not the absence of bugs" [Dijkstra]

Specifications

- To prove that an algorithm correctly solves a problem, we need:
  - A strict problem specification
  - A strict algorithm specification

- A problem can be specified by:
  - *inputs*: input data and associated restrictions (preconditions)
  - *outputs*: output data and associated restrictions (postconditions)

Preconditions and postconditions

double squareRoot(double x)	
preconditions:	x >= 0
postconditions:	`RESULT` * `RESULT` = x (less than a certain error) `RESULT` >= 0

long sum (int n)	
preconditions:	n > 0
postconditions:	`RESULT` = 1 + 2 + … + n

Preconditions and postconditions

template <typename T> void sort(vector<T> v)	
preconditions:	comparison operators defined in T
postconditions:	v[0] <= v[1] <= … <= v[n-1] v has the same elements as initially

template <typename T> T max(vector<T> v)	
preconditions:	v != null comparison operators defined in T
postconditions:	`RESULT` = v[0] or `RESULT` = v[1] or … or `RESULT` = v[n-1] `RESULT` >= v[0] and `RESULT` >= v[1] and … and `RESULT` >= v[n-1]

Partial and total correctness

- **Partial correctness**

  if the algorithm (or program) is executed with inputs that respect the preconditions, then, if ends, it produces correct outputs, i.e., that respect the postconditions

- **Total correctness**

  if the algorithm (or program) is executed with inputs that respect the preconditions, then it ends producing correct outputs, i.e., that respect the postconditions.

Loop invariant

We will focus on the most used algorithmic pattern: a loop

- To prove that a loop is correct, find an invariant of the loop

> *statement about the state of some variable(s) that can be reliable to hold true before and after each iteration of the loop*

and show that

- the invariant is true initially, i.e., is implied by the precondition (*initialization*)
- the invariant is maintained at each iteration, i.e., is true at the end of each iteration, assuming it is true at the beginning of the iteration (*maintenance*)
- when the loop ends, the invariant guarantees (implies) the postcondition (*termination*)

Loop variant

- To prove that a loop ends, find a variant of the loop

> *a non-negative integer expression whose value decreases with each loop execution*

that is, a function (using variables of the loop) that is:

– integer, positive, decreasing

Loop Invariant: Sum (example 1)

```
// returns the sum 1+2+3+…+n

long sum (int n) {
   long sum = 0;
   int k = 1;
   while (k <= n) {
      // invariant: sum = 1+2+…+(k-1)
      sum += k;
      k++;
   }
   return sum;
}
```

Loop invariant: at step *k*, *sum* holds the sum of the the numbers 1 to *k-1*

long sum (int n)	
preconditions:	$n > 0$
postconditions:	RESULT $= 1 + 2 + \ldots + n$

Loop Invariant: Sum (example 1)

Loop invariant: at step *k*, *sum* holds the sum of the numbers 1 to *k-1*

```
public long sum (int n) {
    long sum = 0;
    int k = 1;
    while (k <= n) {
        sum += k;
        k++;
    }
    return sum;
}
```

1. Initialization

the invariant is true at the beginning of the loop

- before the first iteration: k = 1, sum = 0.
- the numbers in range 1 … 1-1 have sum zero (there are no numbers)

→ invariant is true before entering the loop

Loop Invariant: Sum (example 1)

```
public long sum (int n) {
    long sum = 0;
    int k = 1;
    while (k <= n) {
        sum += k;
        k++;
    }
    return sum;
}
```

Loop invariant: at step *k*, *sum* holds the sum of the numbers 1 to *k-1*

2. Maintenance

if invariant is true before step *k*, then it will be true before step *k+1* (immediately after step *k* is finished)

- we assume it is true at beginning of step *k*, "sum is the sum of 1 to *k-1*"
- we have to prove that at the beginning of step *k+1* (after executing step *k*), "sum is the sum of 1 to *k*+1-1", e.g, "sum is the sum of 1 to *k*"
- so, we calculate the value of *sum* at the end of this step:

$sum = 1 + 2 + \ldots + k - 1 + k \quad \rightarrow \quad sum = 1 + 2 + \ldots + k$

Loop Invariant: Sum (example 1)

```
public long sum (int n) {
    long sum = 0;
    int k = 1;
    while (k <= n) {
        sum += k;
        k++;
    }
    return sum;
}
```

Loop invariant: at step *k*, *sum* holds the sum of the numbers 1 to *k-1*

2. Termination

when the loop terminates, the invariant implies the correctness of the algorithm, guarantees the postcondition

– The loop terminates when $k = n+1$

$sum = 1 + 2 + \ldots + n+1-1$

$\rightarrow \quad sum = 1 + 2 + \ldots + n$

Loop variant: Sum (example 1)

```
public long sum (int n) {
   long sum = 0;
   int k = 1;
   while (k <= n) {
      sum += k;
      k++;
   }
   return sum;
}
```

Loop variant: n+1-k

- integer, because *n* and *k* are both integer
- non-negative, because the maximum value of *k* is *n*+1
- decreasing, because k is always incremented

- Algorithm is correct and ends → **Total correctness**

Loop Invariant: Max (example 2)

```
// returns the max value of vector v

int max(const vector<int> v) {
   int max = INT_MIN;
   for (int i = 0; i < v.size() ; i++) {
      // invariant: max = max(v[0], v[1],…, v[i-1])
      if (v[i] > max)
         max  = v[i];
   }
   return max;
}
```

Loop invariant: at step *i*, *max* holds the maximum value of vector *v[0] … v[i-1])*

What is the problem?

- satisfies the “termination” characteristic
- satisfies the “maintenance” characteristic
- does not satisfy the “initialization” characteristic

Loop Invariant: Max (example 2)

```
// returns the max value of vector v

int max(const vector<int> v) {
    if (v.size() == 0)
        throw std::invalid_argument("empty vector");
    int max = v[0];
    for (int i = 1; i < v.size() ; i++) {
        // invariant: max = max(v[0], v[1],…, v[i-1])
        if (v[i] > max)
            max  = v[i];
    }
    return max;
}
```

Loop invariant: at step *k*, *max* holds the maximum value of vector *v*

References

- T.H. Cormen, C. E. Leiserson, R. L. Rivest , C. Stein. Introduction to Algorithms, 3rd Edition. MIT Press, 2009
  - chapter 2 (Getting Started)

- C.A. Furia, B. Meyer, S. Velder, 2014. Loop invariants: Analysis, classification, and examples. *ACM Comput. Surv*. 46(3): Article 34 (January 2014), doi:10.1145/2506375