Algorithms and Data Structures

(1) Correctness of Algorithms

(c) Marcin Sydow

Contact Info

dr hab. Marcin Sydow,

SIAM Department, PJATK

room: 311 (main building)

tel.: +48 22 58 44 571

Organisation

15 lectures + 15 tutorials

tutorials: **total of 60 points (max)**

1. **11 small entry tests** 11 x 2 points = 22 points
2. **2 tests** 2 x 14 points = 28 points
3. **activity, etc.** = max of 10 points

Final mark (tutorials): score divided by 10

(rounded down to the closest mark, but in the range $[2, 5]$)

examples: 36p $\rightarrow$ 3+, 18p $\rightarrow$ 2, 52p $\rightarrow$ 5, etc.

Organisation

15 lectures + 15 tutorials

tutorials: **total of 60 points (max)**

1. **11 small entry tests** 11 x 2 points = 22 points
2. **2 tests** 2 x 14 points = 28 points
3. **activity, etc.** = max of 10 points

Final mark (tutorials): score divided by 10
(rounded down to the closest mark, but in the range $[2, 5]$)

examples: 36p → 3+, 18p → 2, 52p → 5, etc.

exact math formula: grade = $min(5, max(4, \lfloor \frac{score}{5} \rfloor)/2)$

Organisation

15 lectures + 15 tutorials

tutorials: **total of 60 points (max)**

1. **11 small entry tests** 11 x 2 points = 22 points
2. **2 tests** 2 x 14 points = 28 points
3. **activity, etc.** = max of 10 points

Final mark (tutorials): score divided by 10
(rounded down to the closest mark, but in the range $[2, 5]$)

examples: 36p $\rightarrow$ 3+, 18p $\rightarrow$ 2, 52p $\rightarrow$ 5, etc.

exact math formula: grade = $min(5, max(4, \lfloor \frac{score}{5} \rfloor)/2)$

after passing tutorials: **Exam**

(must **pass tutorials to take the exam**)

Books:

General:

- **T.Cormen, C.Leiserson, R.Rivest et al. "Introduction to Algorithms", MIT Press** an excellent textbook for beginners and practitioners (also available in Polish: "Wprowadzenie do Algorytmów, WNT 2000")
- "Algorithms and Datastructures. The Basic Toolbox" (MS), K.Mehlhorn P.Sanders, Springer 2008
- (in Polish) L.Banachowski, K.Diks, W.Rytter "Algorytmy i Struktury Danych", WNT 2001, (290 stron), zwięzła książeczka, trudniejsza dla początkujących
- (Exercises in Polish) G.Mirkowska et al. "Algorytmy i Struktury Danych - Zadania", wydawnictwo PJWSTK, 2005 (zbiór zadań i ćwiczeń, częściowo z rozwiązaniami)

Additional Examples of Books

- N.Wirth “Algorithms + Data Structures = Programs” (also in Polish)
- A.Aho, J.Hopcroft, J.Ullman “Algorithms and Data Structures” (also in Polish)
- (in Polish) W.Lipski “Kombinatoryka dla Programistów”, WNT 2004

For deeper studies:

- D.Knuth “The Art of Computer Programming” 3 volumes, detailed analyses (also in Polish)
- Ch.Papadimitriou “Computational Complexity” more mathematical (also in Polish)

Algorithm

What does “algorithm” mean?

Algorithm

What does "algorithm" mean?

A **recipe** (how to do something, list of actions, etc.)

According to historians the word is derived from the (arabic version of the) name *"al-Khwarizmi"* of a Persian mathematician (A.D. 780-850)

Algorithmics is the heart of computer science
The role of algorithms becomes even more important nowadays (growing data, Internet, search engines, etc.)

1 level above programming languages

Pseudocode

- an abstract notation of algorithm
- looks similar to popular programming languages (Java, C/C++, Pascal)
- plays rather **informative** role than formal (relaxed syntax formalism)

- literals (numbers, strings, null)
- variables (no declarations, but must be initialized)
- arrays ([ ] operator) - we assume that arrays are indexed from 0
- operators (assignment =, comparison (e.g. ==), arithmetic (e.g. +, ++, +=), logic (e.g. !)
- functions (including recursion), the `return` instruction
- conditional statement (IF), loops (FOR, WHILE).

An example of pseudocode usage:

Task: compute sum of numbers in an array of length `len`:

```
sum(array, len){
  sum = 0
  i = 0
  while(i < len){
    sum += array[i]
    i++
  }
  return sum
}
```

(it is not any particular programming language but **precisely expresses the algorithm**

For conveniece, sometimes the '.' (dot) operator will be used (object access operator - the same as in Java, C++, etc.)

For example:

```
if ((node.left != null) && (node.value == 5)) node.updateLeft()
```

What is this course about?

Topics:

1. Algorithm Design
2. Algorithm Analysis
3. Data Structures

Algorithm Design

There is a computational **task** to be performed on computer.

First, the algorithm should be **designed**

Then, the algorithm should be implemented (with some programming language)

Algorithm design (and analysis) is
a **necessary step before programming**

Algorithm Specification

How to express the task "to be done" in algorithmics?

Specification expresses the task. Specification consists of:

- (optional) **name** of algorithm and list of its arguments
- **initial condition** (it specifies what is "correct" **input data** to the problem)
- **final condition** (it specifies what is the desired **result** of the algorithm)

The conditions could be expressed in words, assuming it is precise

Example of a task and its specification

Assuming the task: **"given the array and its length compute the sum of numbers in this array"**

the corresponding **Specification** could be:

name: `sum(Arr, len)`

input: (initial condition)

Algorithm gets 2 following arguments (input data):

1. `Arr` - array of integer numbers
2. `len` - length of `Arr` (natural number)

output:(final condition)

Algorithm must return:

- `sum` - sum of the first `len` numbers in the array `Arr` (integer number)

(any algorithm satisfying the above will be regarded as "correct")

Total Correctness of Algorithm

correct input data is the data which satisfies the **initial condition** of the specification

correct output data is the data which satisfies the **final condition** of the specification

Definition

An algorithm Is called totally correct for the given specification if and only if for any **correct input data** it:

1. **stops** and
2. returns **correct output**

Notice the split into 2 sub-properties in the definition above.

Partial Correctness of Algorithm

Usually, while checking the correctness of an algorithm it is easier to separately:

1. first check whether the algorithm stops
2. then checking the "remaining part". This "remaining part" of correctness is called Partial Correctness of algorithm

Definition

An algorithm is **partially correct** if satisfies the following condition: **If** the algorithm receiving **correct** input data **stops** **then** its result is correct

Note: Partial correctness **does not make** the algorithm stop.

An example of partially correct algorithm

(computing the sum of array of numbers)

```
sum(array, len){
  sum = 0
  i = 0
  while(i < len)
    sum += array[i]
  return sum
}
```

Is this algorithm partially correct?

Is it also totally correct?

The “Stop Property”

A **proof of total correctness** of an algorithm usually assumes **2 separate steps**:

1. (to prove that) the algorithm always stops for correct input data (**stop property**)
2. (to prove that) the algorithm is **partially correct**

(Stop property is usually easier to prove)

Stop property - an example


```
sum(array, len){
  sum = 0
  i = 0
  while(i < len){
    sum += array[i]
    i++
  }
  return sum
}
```

How to easily prove that this algorithm has stop property?

Stop property - an example


```
sum(array, len){
  sum = 0
  i = 0
  while(i < len){
    sum += array[i]
    i++
  }
  return sum
}
```

How to easily prove that this algorithm has stop property? It is enough to observe that:

Stop property - an example

```
sum(array, len){
  sum = 0
  i = 0
  while(i < len){
    sum += array[i]
    i++
  }
  return sum
}
```

How to easily prove that this algorithm has stop property? It is enough to observe that:

1. the algorithm stops when the value of variable i is greater or equal than len

Stop property - an example

```
sum(array, len){
  sum = 0
  i = 0
  while(i < len){
    sum += array[i]
    i++
  }
  return sum
}
```

How to easily prove that this algorithm has stop property? It is enough to observe that:

1. the algorithm stops when the value of variable i is greater or equal than len
2. value of len is a **constant** and finite natural number (according to the specification of this algorithm)

Stop property - an example


```
sum(array, len){
  sum = 0
  i = 0
  while(i < len){
    sum += array[i]
    i++
  }
  return sum
}
```

How to easily prove that this algorithm has stop property? It is enough to observe that:

1. the algorithm stops when the value of variable i is greater or equal than len
2. value of len is a **constant** and finite natural number (according to the specification of this algorithm)
3. value of i increases by 1 with each iteration of the loop

Stop property - an example


```
sum(array, len){
  sum = 0
  i = 0
  while(i < len){
    sum += array[i]
    i++
  }
  return sum
}
```

How to easily prove that this algorithm has stop property? It is enough to observe that:

1. the algorithm stops when the value of variable i is greater or equal than len
2. value of len is a **constant** and finite natural number (according to the specification of this algorithm)
3. value of i increases by 1 with each iteration of the loop

As the result, the algorithm **will certainly stop** after finite number of iterations for any input correct data

Proving Partial Correctness - Invariants

Proving the stop property of an algorithm is usually easy. Proving the "remaining part" of its total correctness (i.e. partial correctness) needs usually more work and sometimes invention, even for quite simple algorithms.

Observation: most of activity of algorithms can be expressed in the form of "WHILE loop". Thus, a **tool** for examining the correctness of loops would be highly useful.

Invariant of a loop is such a tool.

Definition

A loop invariant is a logical predicate such that:
IF it is satisfied **before** entering any single iteration of the loop **THEN** it is also satisfied **after** that iteration.

An example of a typical task in algorithmics:

What does the following algorithm "do" (prove your answer):
(the names of variables are purposely obscure :))
input: Arr - an array of integers, len > 0 - length of array

```
algor1(Arr, len){
  i = 1
  x = Arr[0]
  while(i < len)
    if(Arr[i] > x){
      x = Arr[i]
    }
    i++
  return x
}
```

An example of a typical task in algorithmics:

What does the following algorithm "do" (prove your answer):
(the names of variables are purposely obscure :))
input: Arr - an array of integers, len > 0 - length of array

```
algor1(Arr, len){
  i = 1
  x = Arr[0]
  while(i < len)
    if(Arr[i] > x){
      x = Arr[i]
    }
    i++
  return x
}
```

Easy? OK.

An example of a typical task in algorithmics:

What does the following algorithm “do” (prove your answer):
(the names of variables are purposely obscure :))
input: Arr - an array of integers, len > 0 - length of array

```
algor1(Arr, len){
  i = 1
  x = Arr[0]
  while(i < len)
    if(Arr[i] > x){
      x = Arr[i]
    }
    i++
  return x
}
```

Easy? OK. But now it is also necessary **to prove** the answer.
More precisely, the proof of **total correctness** is needed.

An example - proving total correctness, cont.

2 steps are needed (what steps?)

An example - proving total correctness, cont.

2 steps are needed (what steps?)

1. proving the stop property of algorithm

An example - proving total correctness, cont.

2 steps are needed (what steps?)

1. proving the stop property of algorithm
2. proving the partial correctness of algorithm

An example - proving total correctness, cont.

2 steps are needed (what steps?)

1. proving the stop property of algorithm
2. proving the partial correctness of algorithm

Stop property?

An example - proving total correctness, cont.

2 steps are needed (what steps?)

1. proving the stop property of algorithm
2. proving the partial correctness of algorithm

Stop property?

```
algor1(Arr, len){
  i = 1
  x = Arr[0]
  while(i < len)
    if(Arr[i] > x){
      x = Arr[i]
    }
    i++
  return x
}
```

An example - proving total correctness, cont.

2 steps are needed (what steps?)

1. proving the stop property of algorithm
2. proving the partial correctness of algorithm

Stop property?

```
algor1(Arr, len){
  i = 1
  x = Arr[0]
  while(i < len)
    if(Arr[i] > x){
      x = Arr[i]
    }
    i++
  return x
}
```

It was easy.

An example - proving total correctness, cont.

2 steps are needed (what steps?)

1. proving the stop property of algorithm
2. proving the partial correctness of algorithm

Stop property?

```
algor1(Arr, len){
  i = 1
  x = Arr[0]
  while(i < len)
    if(Arr[i] > x){
      x = Arr[i]
    }
    i++
  return x
}
```

It was easy. Now, partial correctness...

Example continued - partial correctness

```
algor1(Arr, len){
  i = 1
  x = Arr[0]
  while(i < len)
    if(Arr[i] > x){
      x = Arr[i]
    }
    i++
  return x
}
```

we would like to show that “x is a maximum in Arr”

Example continued - partial correctness


```
algor1(Arr, len){
  i = 1
  x = Arr[0]
  while(i < len)
    if(Arr[i] > x){
      x = Arr[i]
    }
    i++
  return x
}
```

we would like to show that "x is a maximum in Arr" in mathematical notation it would look like:

Example continued - partial correctness


```
algor1(Arr, len){
  i = 1
  x = Arr[0]
  while(i < len)
    if(Arr[i] > x){
      x = Arr[i]
    }
    i++
  return x
}
```

we would like to show that “x is a maximum in Arr” in mathematical notation it would look like:

$(\forall_{0 \leq j < len}\, x \geq Arr[j]) \wedge (\exists_{0 \leq j < len}(x == Arr[j]))$

Example continued - partial correctness


```
algor1(Arr, len){
  i = 1
  x = Arr[0]
  while(i < len)
    if(Arr[i] > x){
      x = Arr[i]
    }
    i++
  return x
}
```

we would like to show that "x is a maximum in Arr" in mathematical notation it would look like:

$(\forall_{0 \leq j < len}\, x \geq Arr[j]) \wedge (\exists_{0 \leq j < len}(x == Arr[j]))$

Ok, but how to show the partial correctness of this algorithm?

Example continued - partial correctness


```
algor1(Arr, len){
  i = 1
  x = Arr[0]
  while(i < len)
    if(Arr[i] > x){
      x = Arr[i]
    }
    i++
  return x
}
```

we would like to show that "x is a maximum in Arr" in mathematical notation it would look like:
$(\forall_{0 \leq j < len} \, x \geq Arr[j]) \wedge (\exists_{0 \leq j < len}(x == Arr[j]))$

Ok, but how to show the partial correctness of this algorithm?

Answer: we can use a loop invariant.

Example continued - application of invariant

Target: $(\forall_{0 \leq j < len}\, x \geq Arr[j]) \wedge (\exists_{0 \leq j < len}(x == Arr[j]))$

```
algor1(Arr, len){
  i = 1
  x = Arr[0]
  while(i < len)
    if(Arr[i] > x){
      x = Arr[i]
    }
    i++
  return x
}
```

Example continued - application of invariant

Target: $(\forall_{0 \leq j < len}\, x \geq Arr[j]) \wedge (\exists_{0 \leq j < len}(x == Arr[j]))$

```
algor1(Arr, len){
  i = 1
  x = Arr[0]
  while(i < len)
    if(Arr[i] > x){
      x = Arr[i]
    }
    i++
  return x
}
```

Invariant: $\forall_{0 \leq j < i}\, x \geq Arr[j] \wedge (\exists_{0 \leq j < len}(x == Arr[j]))$

Example continued - application of invariant

Target: $(\forall_{0 \leq j < len}\, x \geq Arr[j]) \wedge (\exists_{0 \leq j < len}(x == Arr[j]))$

```
algor1(Arr, len){
  i = 1
  x = Arr[0]
  while(i < len)
    if(Arr[i] > x){
      x = Arr[i]
    }
    i++
  return x
}
```

Invariant: $\forall_{0 \leq j < i}\, x \geq Arr[j] \wedge (\exists_{0 \leq j < len}(x == Arr[j]))$

What do we get?

Example continued - application of invariant

Target: $(\forall_{0 \leq j < len}\, x \geq Arr[j]) \wedge (\exists_{0 \leq j < len}(x == Arr[j]))$

```
algor1(Arr, len){
  i = 1
  x = Arr[0]
  while(i < len)
    if(Arr[i] > x){
      x = Arr[i]
    }
    i++
  return x
}
```

Invariant: $\forall_{0 \leq j < i}\, x \geq Arr[j] \wedge (\exists_{0 \leq j < len}(x == Arr[j]))$
What do we get? In **conjuction** with the **stop condition of the loop**

Example continued - application of invariant

Target: $(\forall_{0 \leq j < len}\, x \geq Arr[j]) \wedge (\exists_{0 \leq j < len}(x == Arr[j]))$

```
algor1(Arr, len){
  i = 1
  x = Arr[0]
  while(i < len)
    if(Arr[i] > x){
      x = Arr[i]
    }
    i++
  return x
}
```

Invariant: $\forall_{0 \leq j < i}\, x \geq Arr[j] \wedge (\exists_{0 \leq j < len}(x == Arr[j]))$
What do we get? In **conjuction** with the **stop condition of the loop** $(i == len)$

Example continued - application of invariant

Target: $(\forall_{0 \leq j < len}\, x \geq Arr[j]) \wedge (\exists_{0 \leq j < len}(x == Arr[j]))$

```
algor1(Arr, len){
  i = 1
  x = Arr[0]
  while(i < len)
    if(Arr[i] > x){
      x = Arr[i]
    }
    i++
  return x
}
```

Invariant: $\forall_{0 \leq j < i}\, x \geq Arr[j] \wedge (\exists_{0 \leq j < len}(x == Arr[j]))$
What do we get? In **conjuction** with the **stop condition of the loop** ($i == len$) we got the proof!
$((\forall_{0 \leq j < i}\, x \geq Arr[j]) \wedge (i == len))$

Example continued - application of invariant

Target: $(\forall_{0 \leq j < len}\, x \geq Arr[j]) \wedge (\exists_{0 \leq j < len}(x == Arr[j]))$

```
algor1(Arr, len){
  i = 1
  x = Arr[0]
  while(i < len)
    if(Arr[i] > x){
      x = Arr[i]
    }
    i++
  return x
}
```

Invariant: $\forall_{0 \leq j < i}\, x \geq Arr[j] \wedge (\exists_{0 \leq j < len}(x == Arr[j]))$
What do we get? In **conjuction** with the **stop condition of the loop** $(i == len)$ we got the proof!
$((\forall_{0 \leq j < i}\, x \geq Arr[j]) \wedge (i == len)) \Rightarrow (\forall_{0 \leq j < len}\, x \geq Arr[j])$

What you should know after this lecture:

1. Organisation and Passing Rules of this course :)
2. What is specification
3. What does “correct input data” mean
4. Definition of Total Correctness of algorithm
5. Definition of Partial Correctness of algorithm
6. What is stop property of an algorithm
7. Be able to give example of a partially correct algorithm which is not totally correct
8. Be able to prove stop property of simple algorithms
9. Definition of invariant of a loop
10. Be able to invent good invariant for a given loop
11. Be able to prove total correctness for simple algorithms

Thank you for your attention