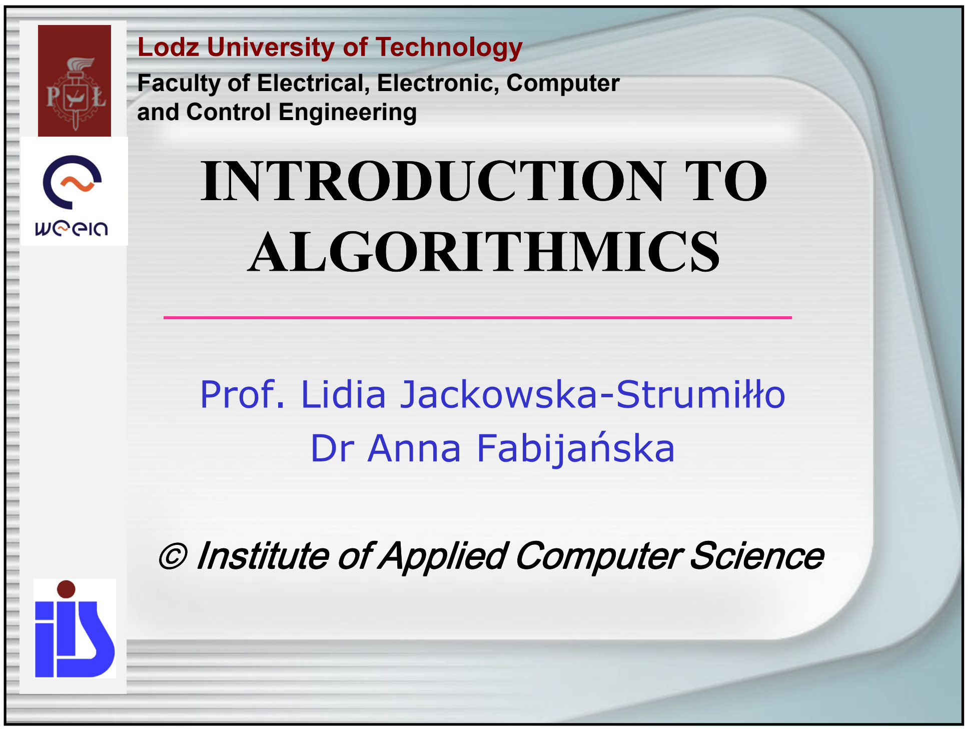
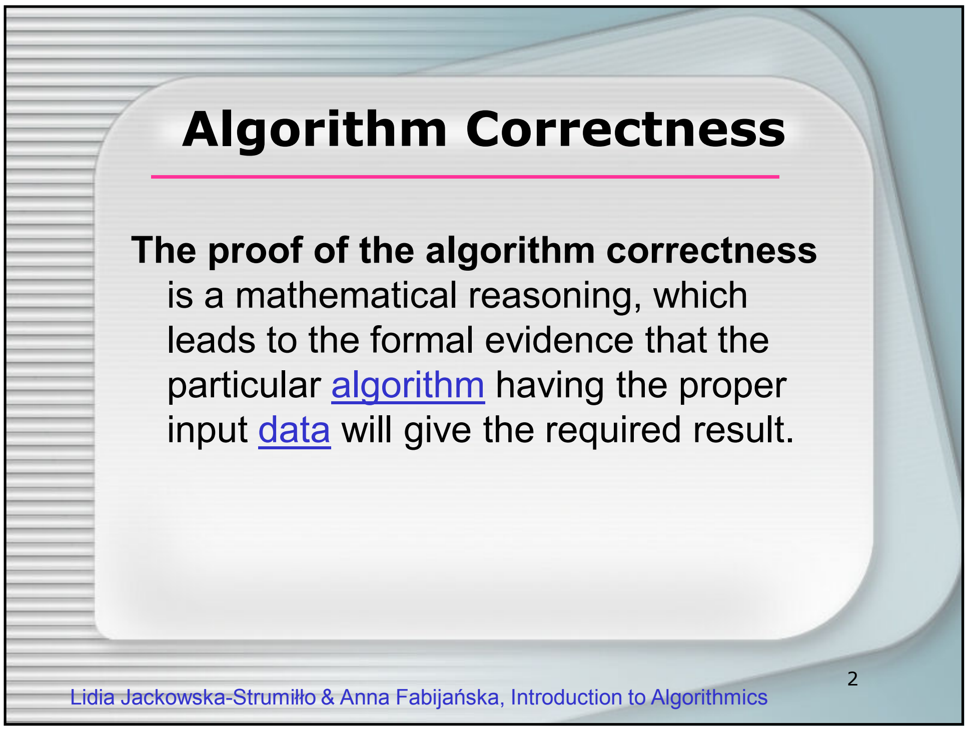


Lodz University of Technology
Faculty of Electrical, Electronic, Computer and Control Engineering

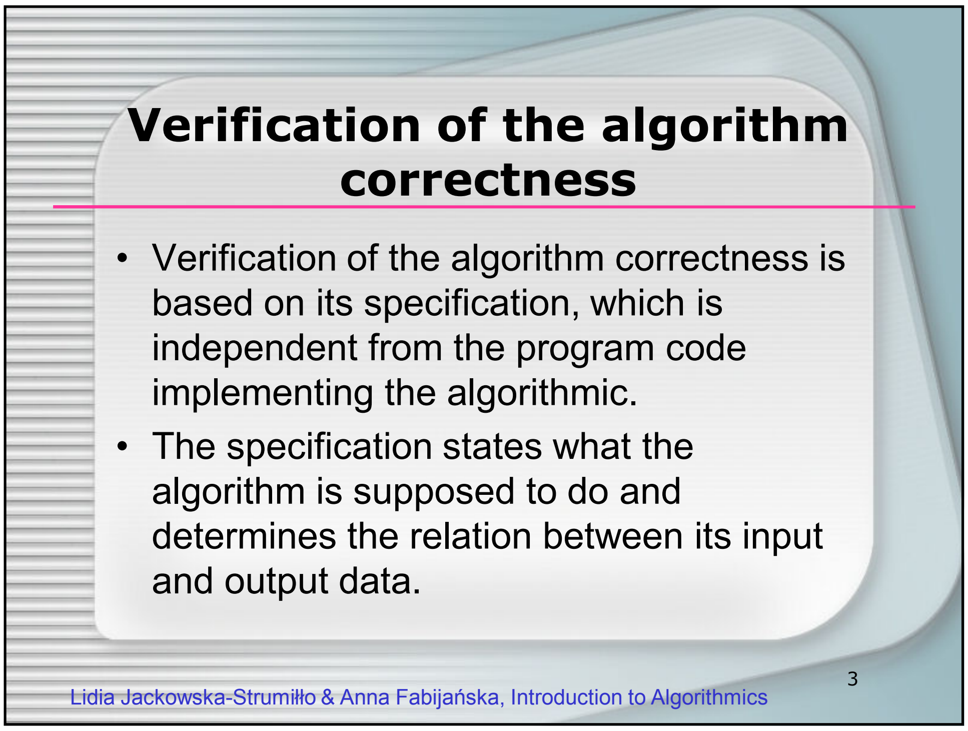
INTRODUCTION TO ALGORITHMICS

Prof. Lidia Jackowska-Strumiłło
Dr Anna Fabijańska



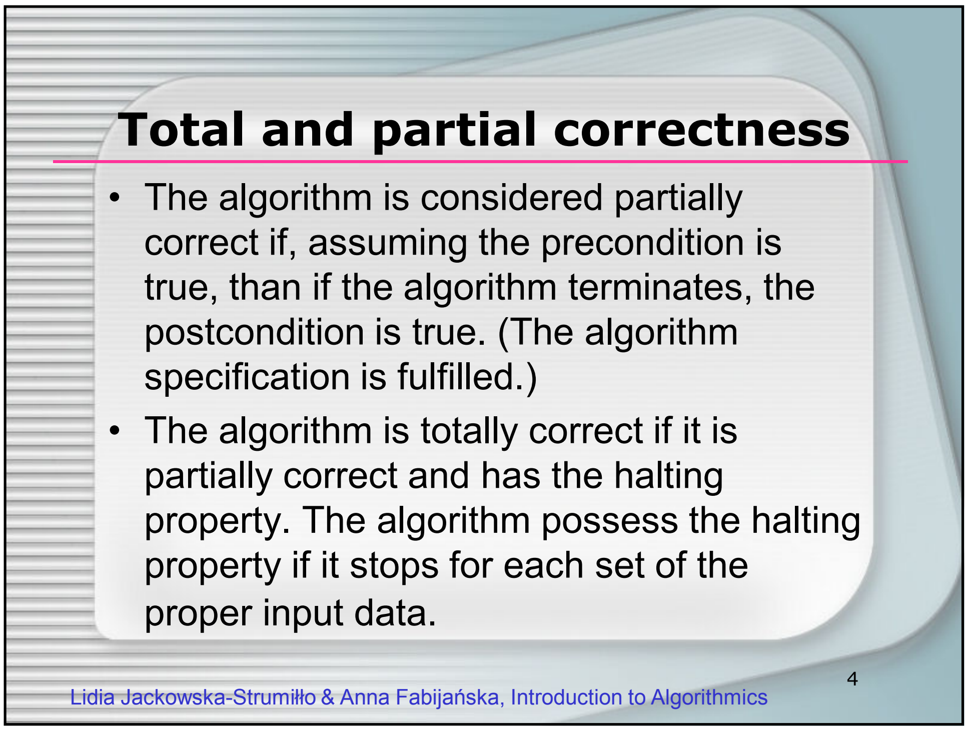
Algorithm Correctness

The proof of the algorithm correctness is a mathematical reasoning, which leads to the formal evidence that the particular algorithm having the proper input data will give the required result.



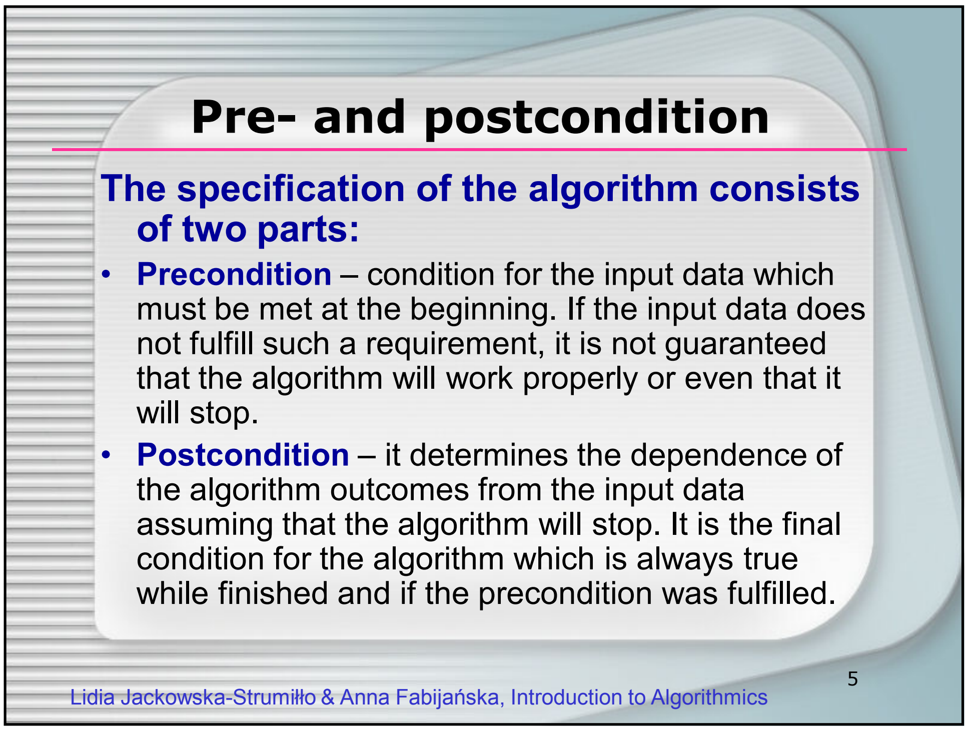
Verification of the algorithm correctness

- Verification of the algorithm correctness is based on its specification, which is independent from the program code implementing the algorithmic.
- The specification states what the algorithm is supposed to do and determines the relation between its input and output data.



Total and partial correctness

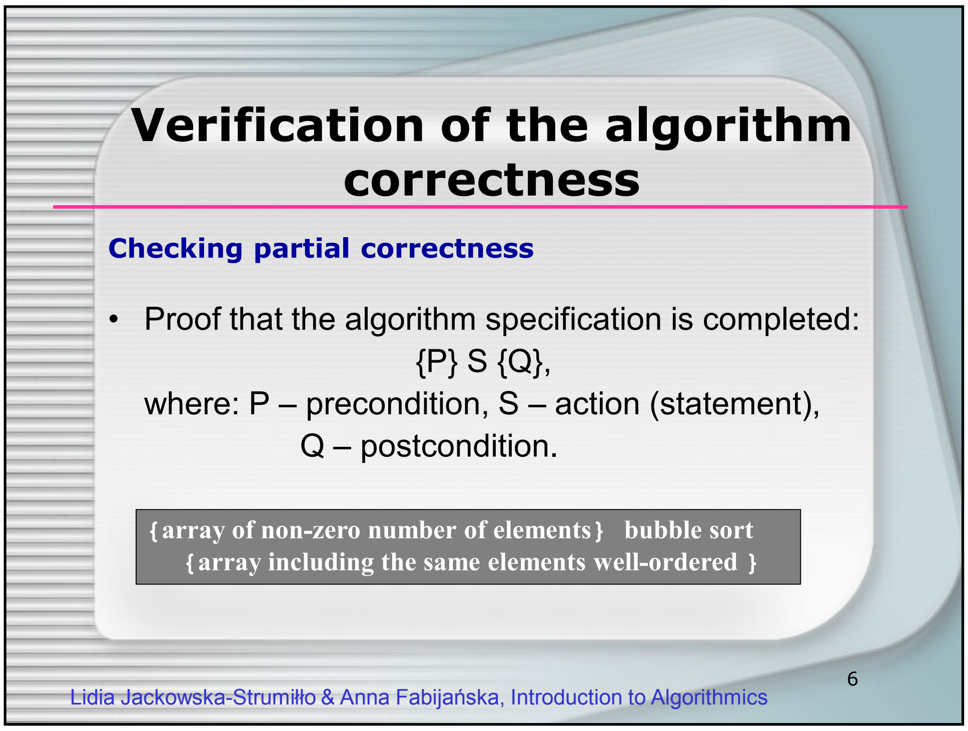
- The algorithm is considered partially correct if, assuming the precondition is true, than if the algorithm terminates, the postcondition is true. (The algorithm specification is fulfilled.)
- The algorithm is totally correct if it is partially correct and has the halting property. The algorithm possess the halting property if it stops for each set of the proper input data.



Pre- and postcondition

The specification of the algorithm consists of two parts:

- **Precondition** – condition for the input data which must be met at the beginning. If the input data does not fulfill such a requirement, it is not guaranteed that the algorithm will work properly or even that it will stop.
- **Postcondition** – it determines the dependence of the algorithm outcomes from the input data assuming that the algorithm will stop. It is the final condition for the algorithm which is always true while finished and if the precondition was fulfilled.



Verification of the algorithm correctness

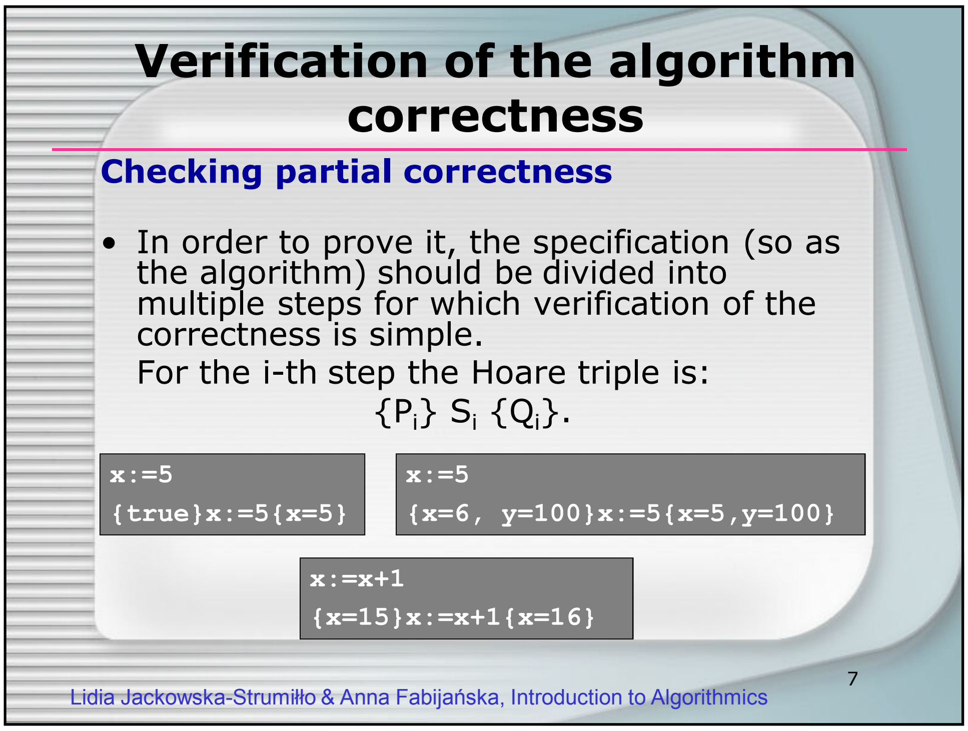
Checking partial correctness

- Proof that the algorithm specification is completed:

  {P} S {Q},

  where: P – precondition, S – action (statement), Q – postcondition.

{array of non-zero number of elements} bubble sort {array including the same elements well-ordered }



Verification of the algorithm correctness

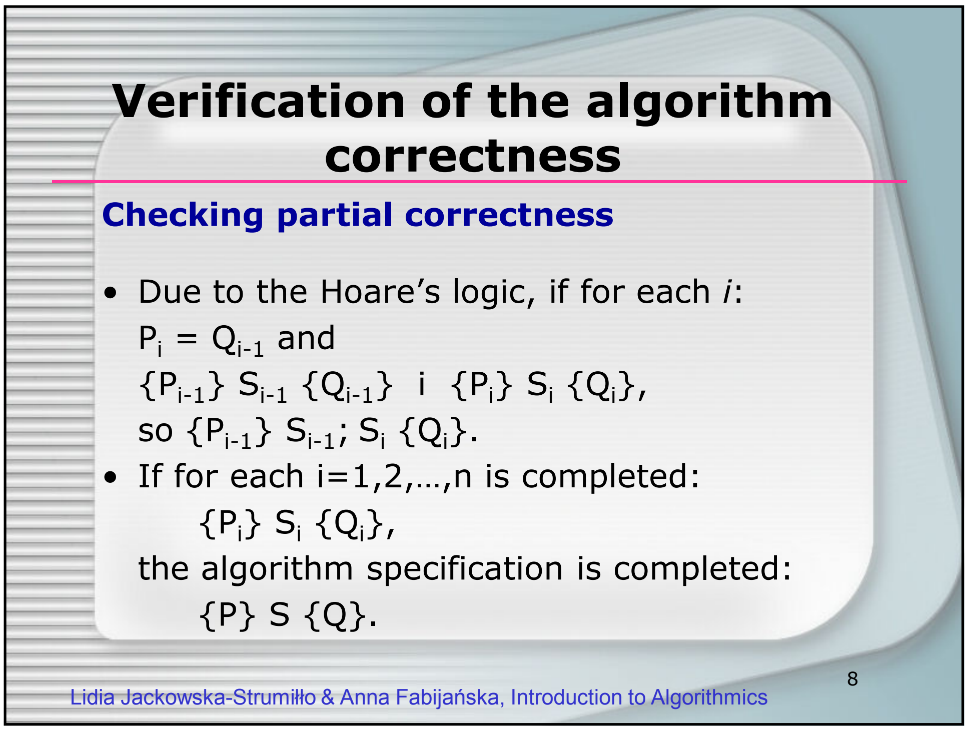
Checking partial correctness

- In order to prove it, the specification (so as the algorithm) should be divided into multiple steps for which verification of the correctness is simple.
  For the i-th step the Hoare triple is:
  $\{P_i\}\ S_i\ \{Q_i\}$.

```
x:=5
{true}x:=5{x=5}
```

```
x:=5
{x=6, y=100}x:=5{x=5,y=100}
```

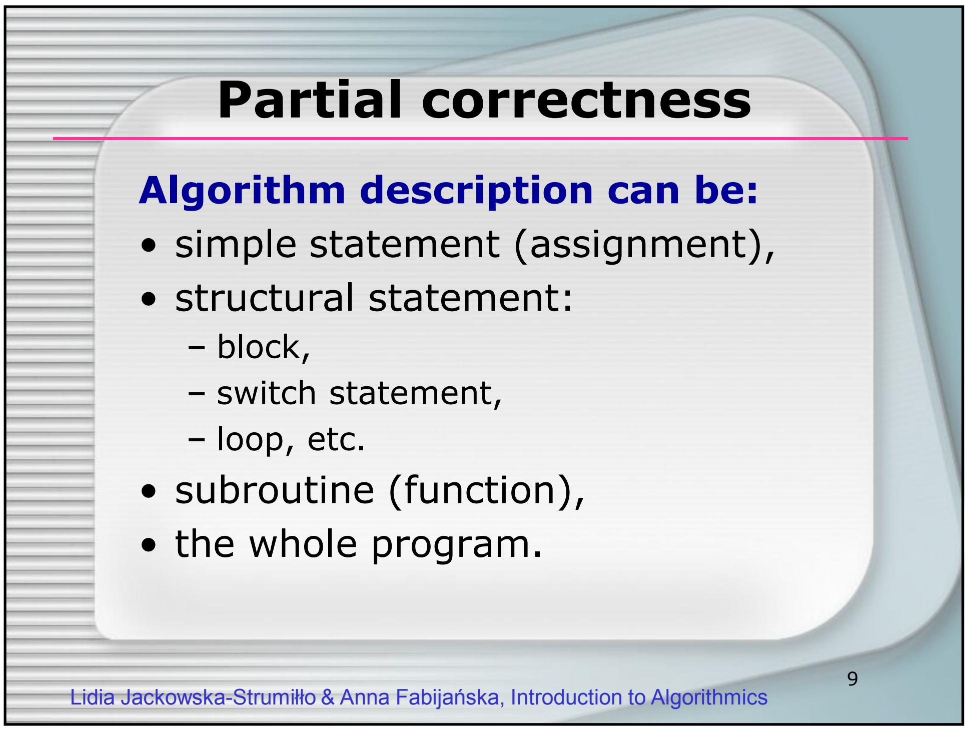
```
x:=x+1
{x=15}x:=x+1{x=16}
```



Verification of the algorithm correctness

Checking partial correctness

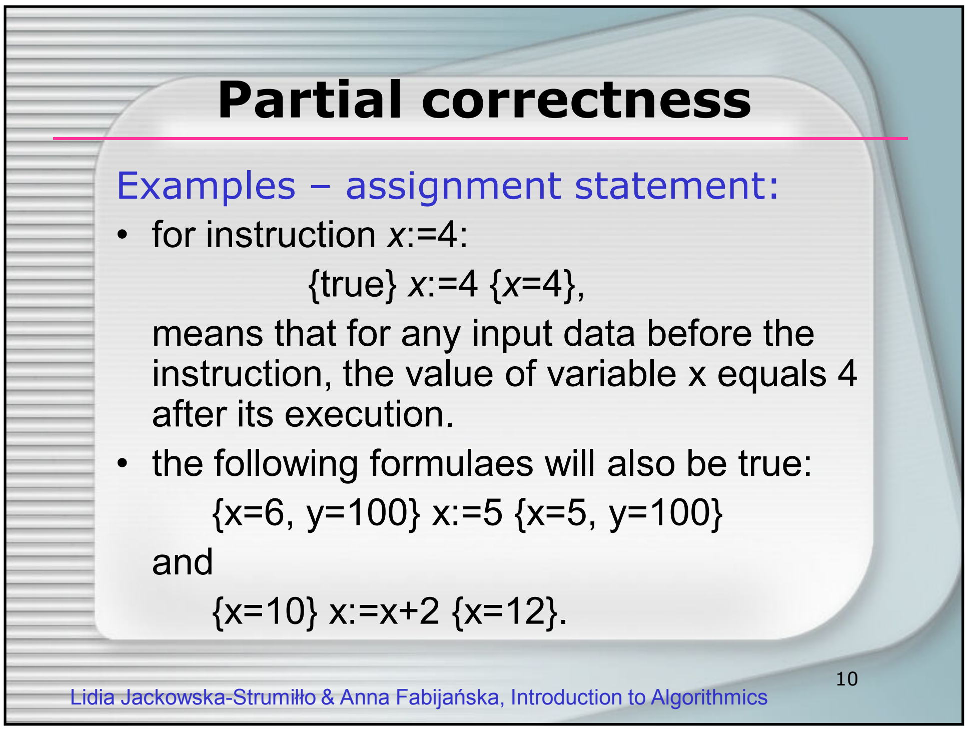
- Due to the Hoare's logic, if for each *i*:
  $P_i = Q_{i-1}$ and
  $\{P_{i-1}\}\ S_{i-1}\ \{Q_{i-1}\}$ i $\{P_i\}\ S_i\ \{Q_i\}$,
  so $\{P_{i-1}\}\ S_{i-1};\ S_i\ \{Q_i\}$.
- If for each i=1,2,…,n is completed:
  $\{P_i\}\ S_i\ \{Q_i\}$,
  the algorithm specification is completed:
  $\{P\}\ S\ \{Q\}$.



Partial correctness

Algorithm description can be:

- simple statement (assignment),
- structural statement:
  - block,
  - switch statement,
  - loop, etc.
- subroutine (function),
- the whole program.



Partial correctness

Examples – assignment statement:

- for instruction *x*:=4:

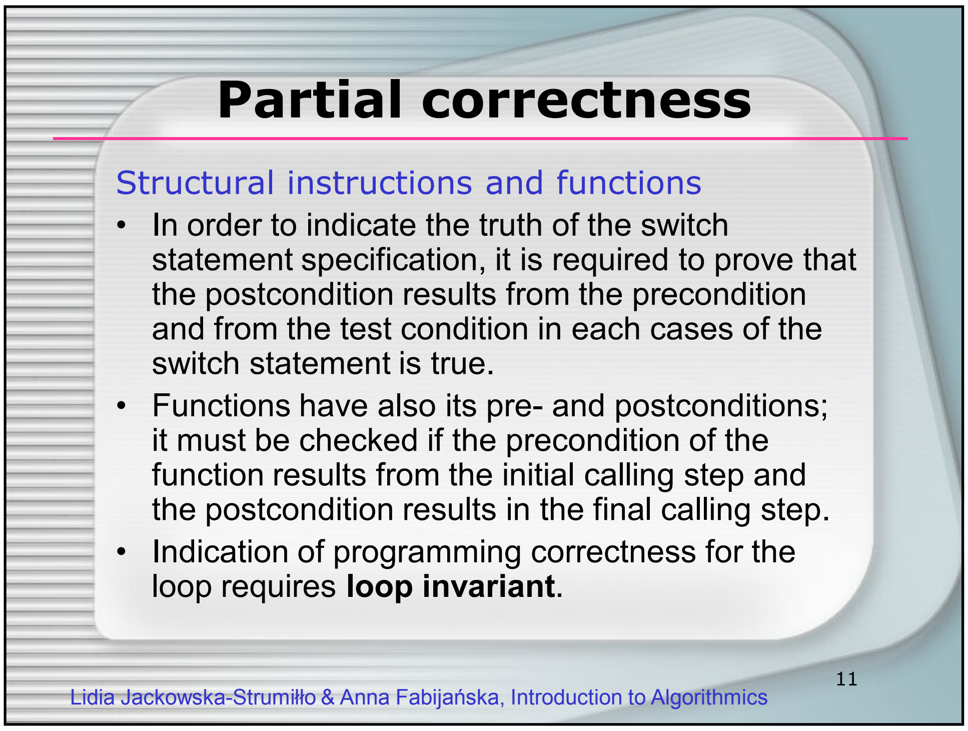
  {true} *x*:=4 {*x*=4},

  means that for any input data before the instruction, the value of variable x equals 4 after its execution.
- the following formulaes will also be true:

  {x=6, y=100} x:=5 {x=5, y=100}

  and

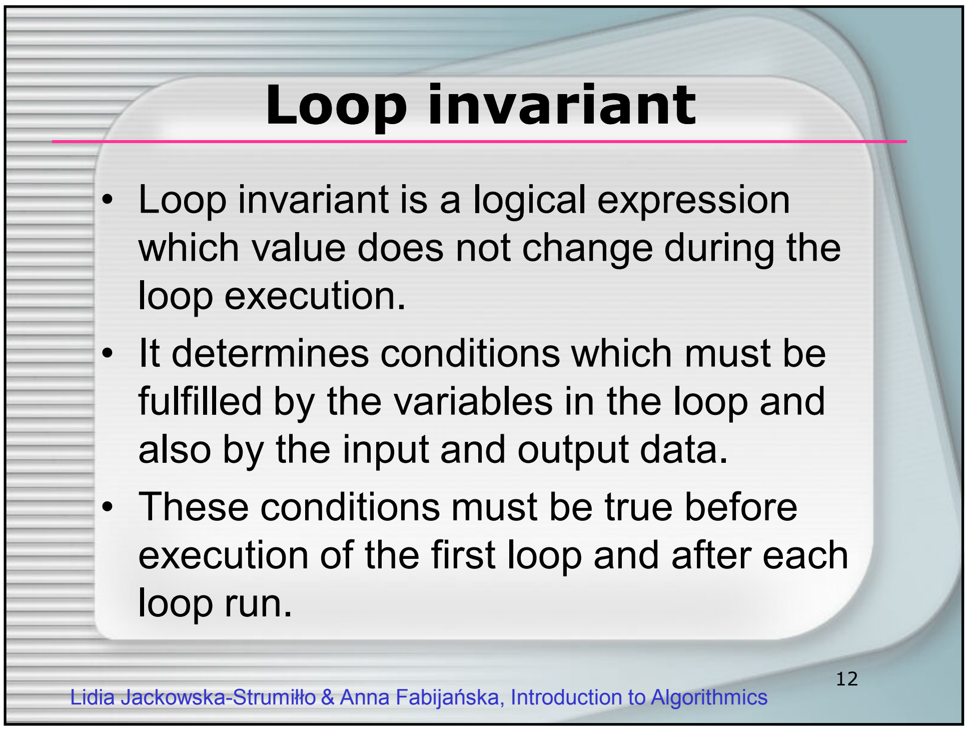
  {x=10} x:=x+2 {x=12}.



Partial correctness

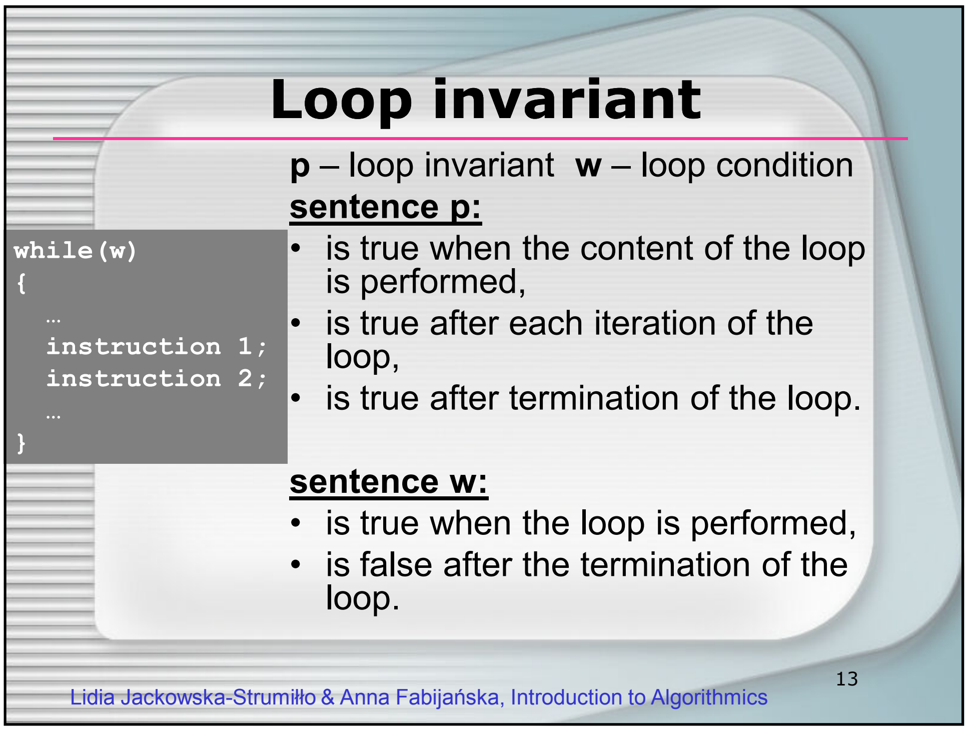
Structural instructions and functions

- In order to indicate the truth of the switch statement specification, it is required to prove that the postcondition results from the precondition and from the test condition in each cases of the switch statement is true.
- Functions have also its pre- and postconditions; it must be checked if the precondition of the function results from the initial calling step and the postcondition results in the final calling step.
- Indication of programming correctness for the loop requires **loop invariant**.



Loop invariant

- Loop invariant is a logical expression which value does not change during the loop execution.
- It determines conditions which must be fulfilled by the variables in the loop and also by the input and output data.
- These conditions must be true before execution of the first loop and after each loop run.



Loop invariant

```
while(w)
{
  …
  instruction 1;
  instruction 2;
  …
}
```

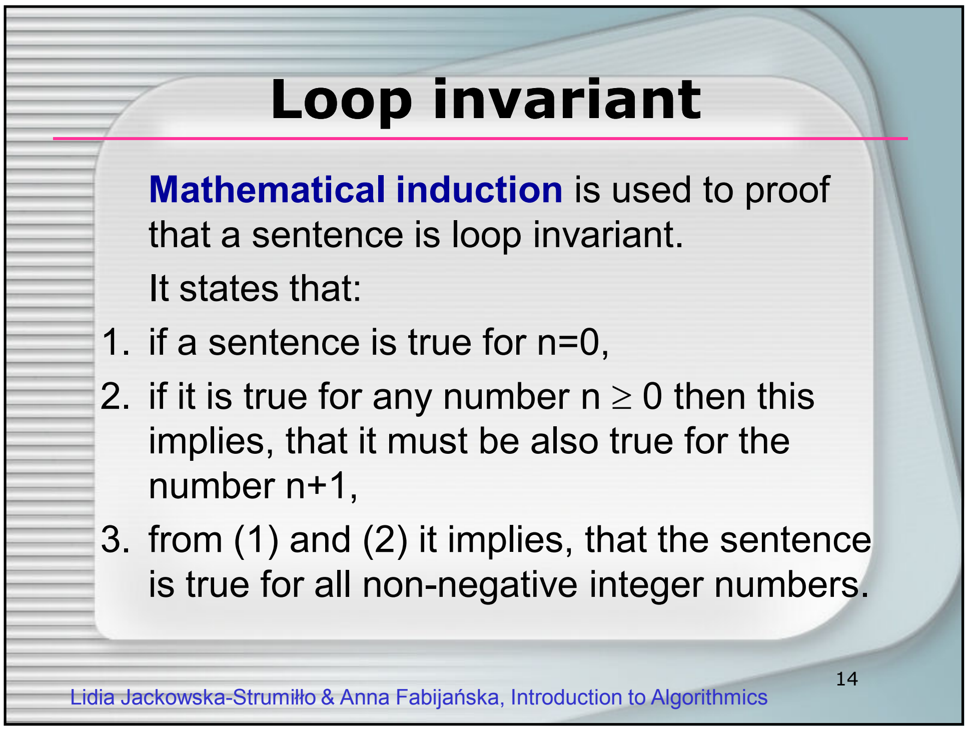
p – loop invariant **w** – loop condition

sentence p:

- is true when the content of the loop is performed,
- is true after each iteration of the loop,
- is true after termination of the loop.

sentence w:

- is true when the loop is performed,
- is false after the termination of the loop.

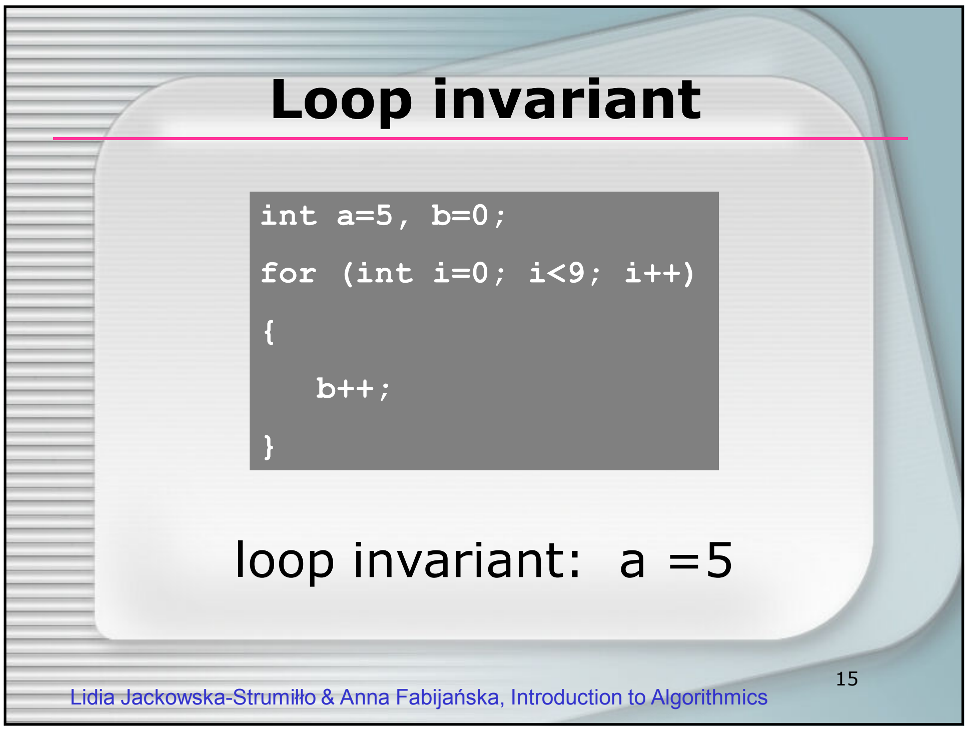


Loop invariant

Mathematical induction is used to proof that a sentence is loop invariant.

It states that:

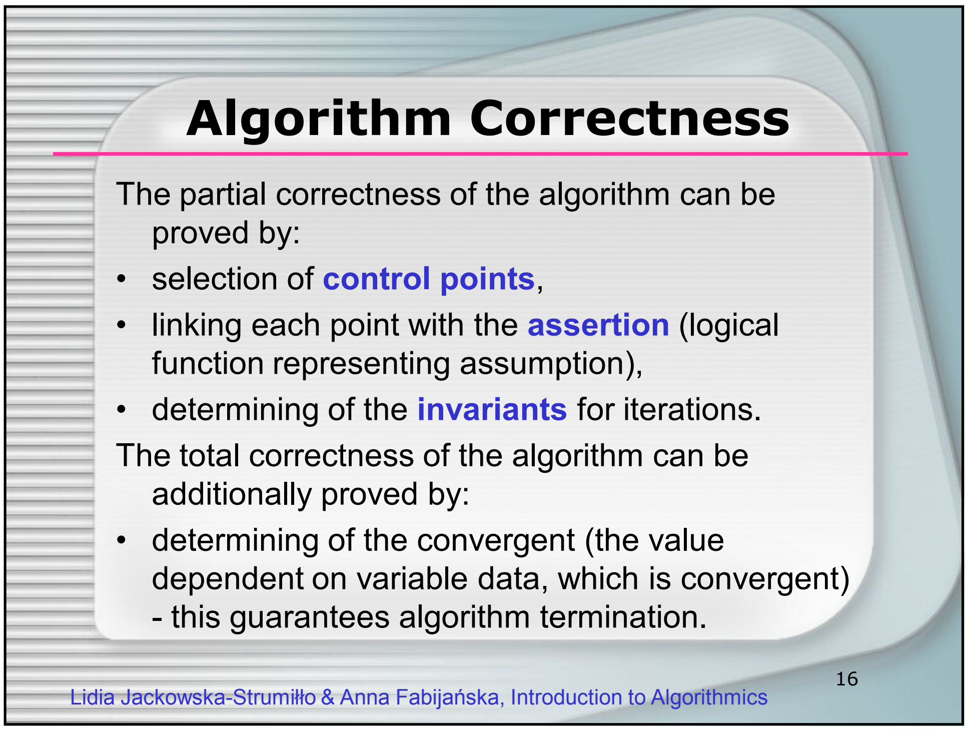
1. if a sentence is true for $n=0$,
2. if it is true for any number $n \geq 0$ then this implies, that it must be also true for the number $n+1$,
3. from (1) and (2) it implies, that the sentence is true for all non-negative integer numbers.



Loop invariant

```
int a=5, b=0;
for (int i=0; i<9; i++)
{
   b++;
}
```

loop invariant: a =5



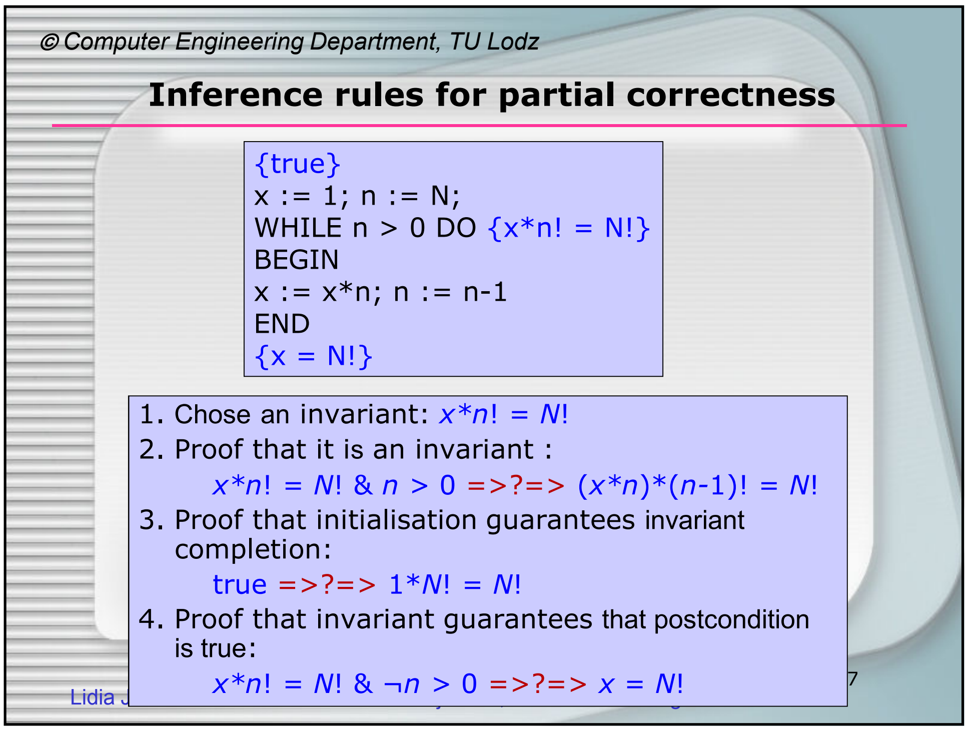
Algorithm Correctness

The partial correctness of the algorithm can be proved by:

- selection of **control points**,
- linking each point with the **assertion** (logical function representing assumption),
- determining of the **invariants** for iterations.

The total correctness of the algorithm can be additionally proved by:

- determining of the convergent (the value dependent on variable data, which is convergent) - this guarantees algorithm termination.



Inference rules for partial correctness

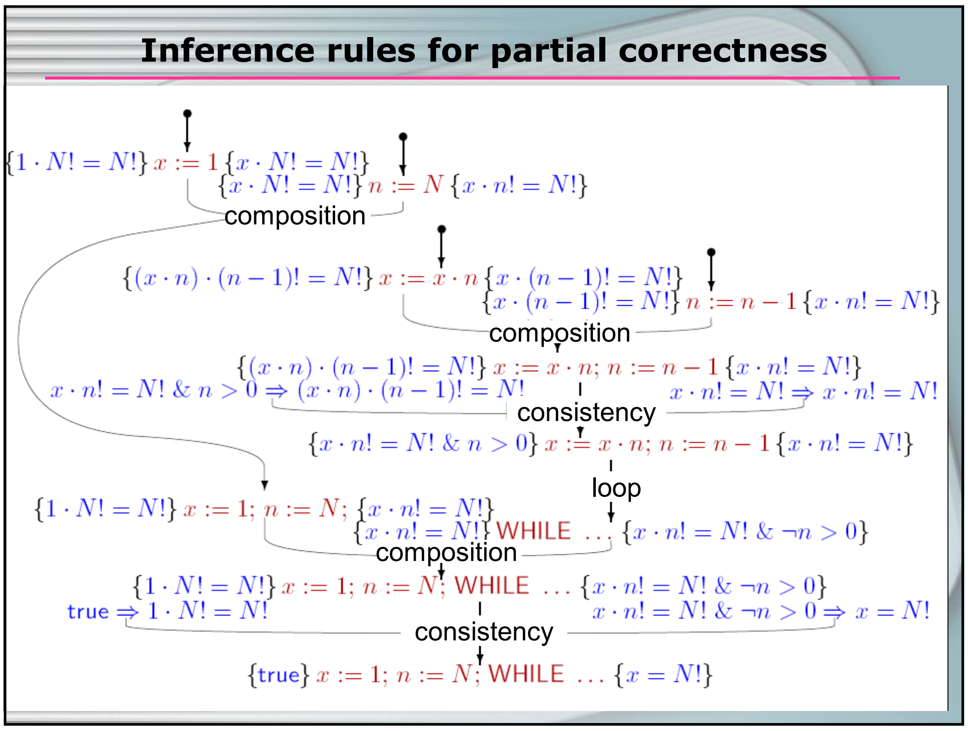
```
{true}
x := 1; n := N;
WHILE n > 0 DO {x*n! = N!}
BEGIN
x := x*n; n := n-1
END
{x = N!}
```

1. Chose an invariant: *x*n*! = *N*!
2. Proof that it is an invariant :

   *x*n*! = *N*! & *n* > 0 =>?=> (*x*n*)*(*n*-1)! = *N*!
3. Proof that initialisation guarantees invariant completion:

   true =>?=> 1**N*! = *N*!
4. Proof that invariant guarantees that postcondition is true:

   *x*n*! = *N*! & ¬*n* > 0 =>?=> *x* = *N*!



Inference rules for partial correctness

$\{1 \cdot N! = N!\}\ x := 1\ \{x \cdot N! = N!\}$ $\{x \cdot N! = N!\}\ n := N\ \{x \cdot n! = N!\}$

composition

$\{(x \cdot n) \cdot (n-1)! = N!\}\ x := x \cdot n\ \{x \cdot (n-1)! = N!\}$ $\{x \cdot (n-1)! = N!\}\ n := n-1\ \{x \cdot n! = N!\}$

composition

$x \cdot n! = N!\ \&\ n > 0 \Rightarrow (x \cdot n) \cdot (n-1)! = N!$ $\{(x \cdot n) \cdot (n-1)! = N!\}\ x := x \cdot n;\ n := n-1\ \{x \cdot n! = N!\}$ $x \cdot n! = N! \Rightarrow x \cdot n! = N!$

consistency

$\{x \cdot n! = N!\ \&\ n > 0\}\ x := x \cdot n;\ n := n-1\ \{x \cdot n! = N!\}$

loop

$\{1 \cdot N! = N!\}\ x := 1;\ n := N;\ \{x \cdot n! = N!\}$ $\{x \cdot n! = N!\}$ WHILE $\ldots$ $\{x \cdot n! = N!\ \&\ \neg n > 0\}$

composition

$\text{true} \Rightarrow 1 \cdot N! = N!$ $\{1 \cdot N! = N!\}\ x := 1;\ n := N;$ WHILE $\ldots$ $\{x \cdot n! = N!\ \&\ \neg n > 0\}$ $x \cdot n! = N!\ \&\ \neg n > 0 \Rightarrow x = N!$

consistency

$\{\text{true}\}\ x := 1;\ n := N;$ WHILE $\ldots$ $\{x = N!\}$

Inference rules for partial correctness

Hoare's logic for partial correctness is:

- **correct** – works out only the true facts and,
- **complete** – each true statement which is partially correct can be conducted.

Non-automatic steps in the proof of partial correctness:

- to guess the loop invariant,
- to proof non-informatic facts required in the consistency rule.

It is possible (with some difficulties) to generalise the Hoare's logic into more complex programming languages.

To proof partial correctness is time consuming as for daily programming practice.

Hoare's logic has influenced the progress of programming languages and methods.

Loop invariant

```
int power1(int x, int n)
{
  int z,m,y;
  z=x; y=1; m=n;
  while (m!=0)
  {
     if (m%2==1) y=y*z;
     m=m/2;
     z=z*z;
  }
  return y;
}
```

binary power

Invariant: $z^m * y = x^n$

n – natural number,
x- real number

$z^m * y = x^n$

$z^{2*m/2} * (y * z) = x^n$ and $m \% 2=1$
or $z^{2*m/2} * y=x^n$ and $m \% 2 = 0$

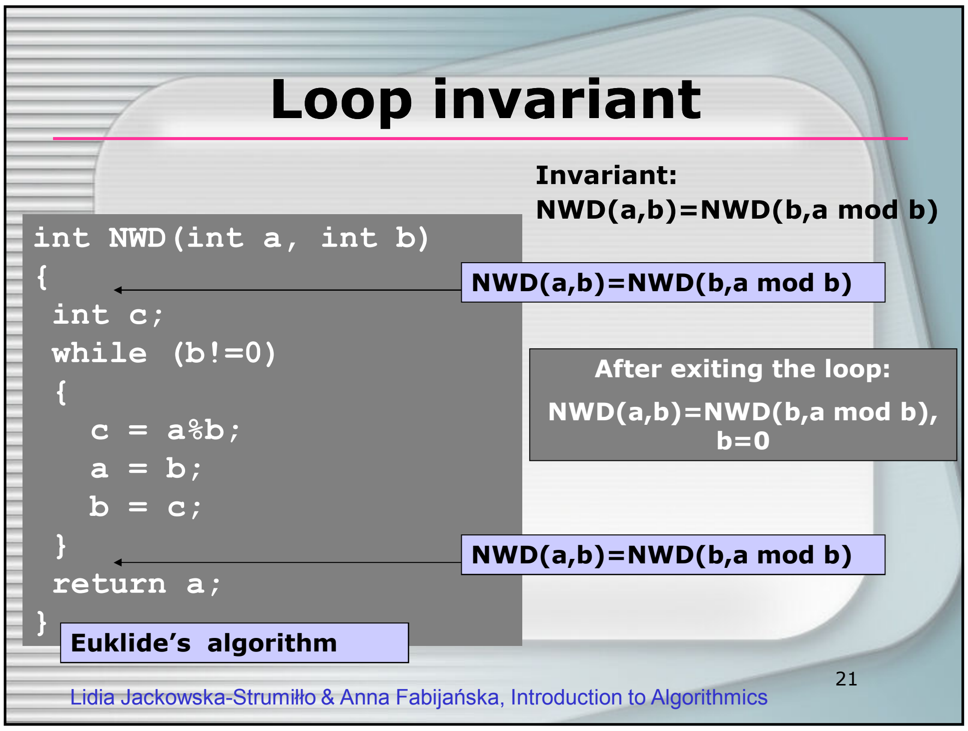
$(z*z)^{m/2} * y = x^n$

$(z*z)^m * y = x^n$

$z^m * y = x^n$

After exiting the loop:

$z^m * y = x^n$, $m=0$ so $y = x^n$



Loop invariant

Invariant:
NWD(a,b)=NWD(b,a mod b)

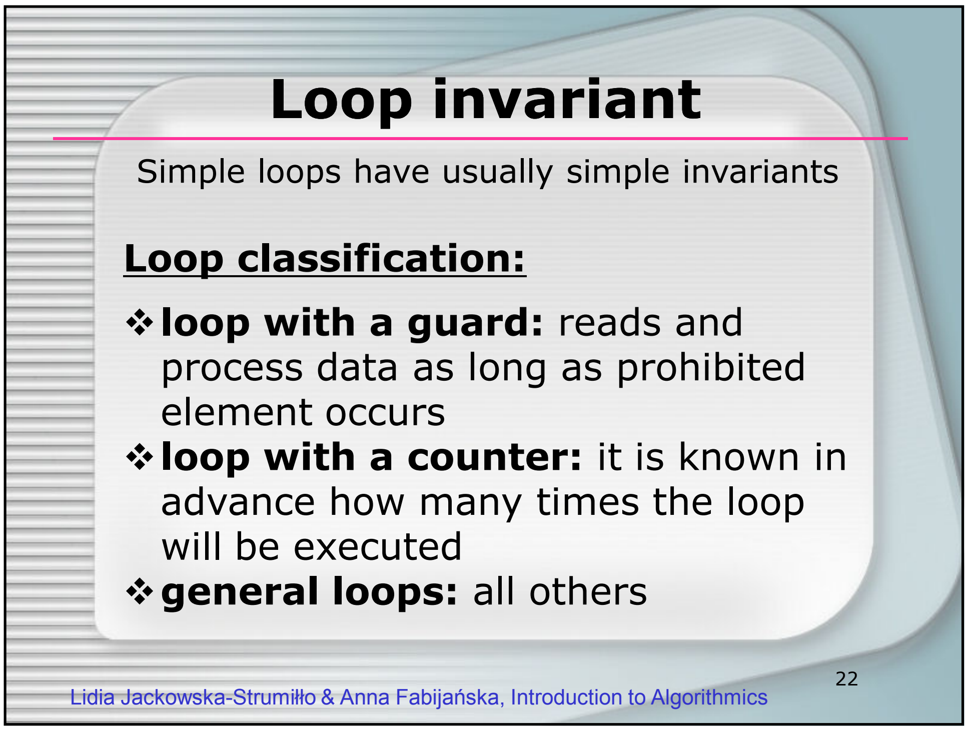
```
int NWD(int a, int b)
{
 int c;
 while (b!=0)
 {
   c = a%b;
   a = b;
   b = c;
 }
 return a;
}
```

NWD(a,b)=NWD(b,a mod b)

After exiting the loop:
NWD(a,b)=NWD(b,a mod b), b=0

NWD(a,b)=NWD(b,a mod b)

Euklide's algorithm

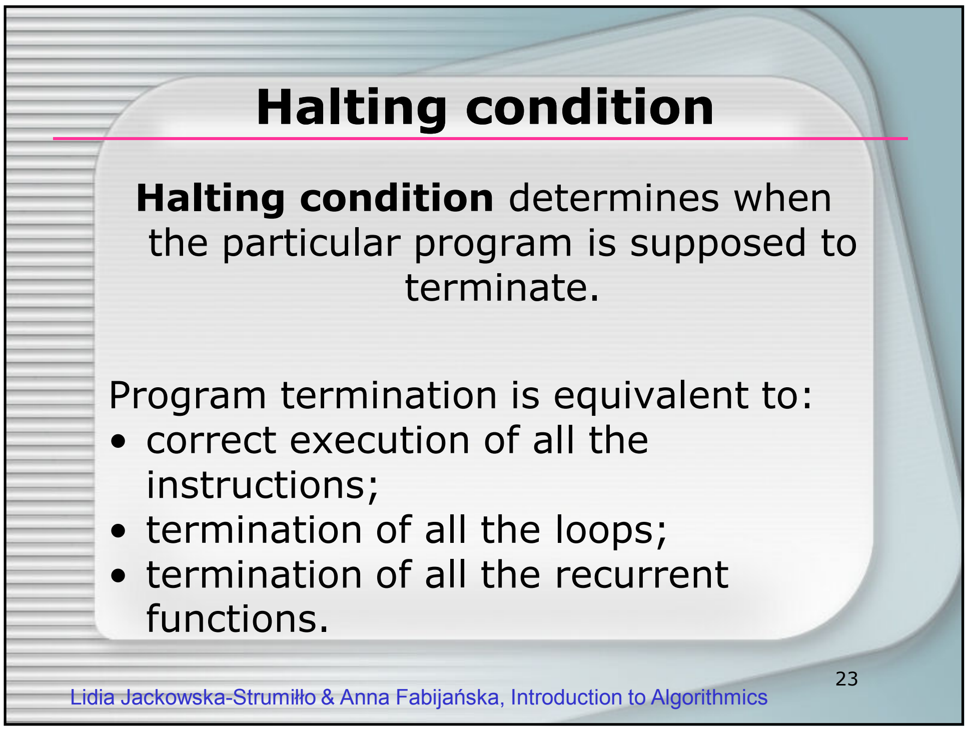


Loop invariant

Simple loops have usually simple invariants

Loop classification:

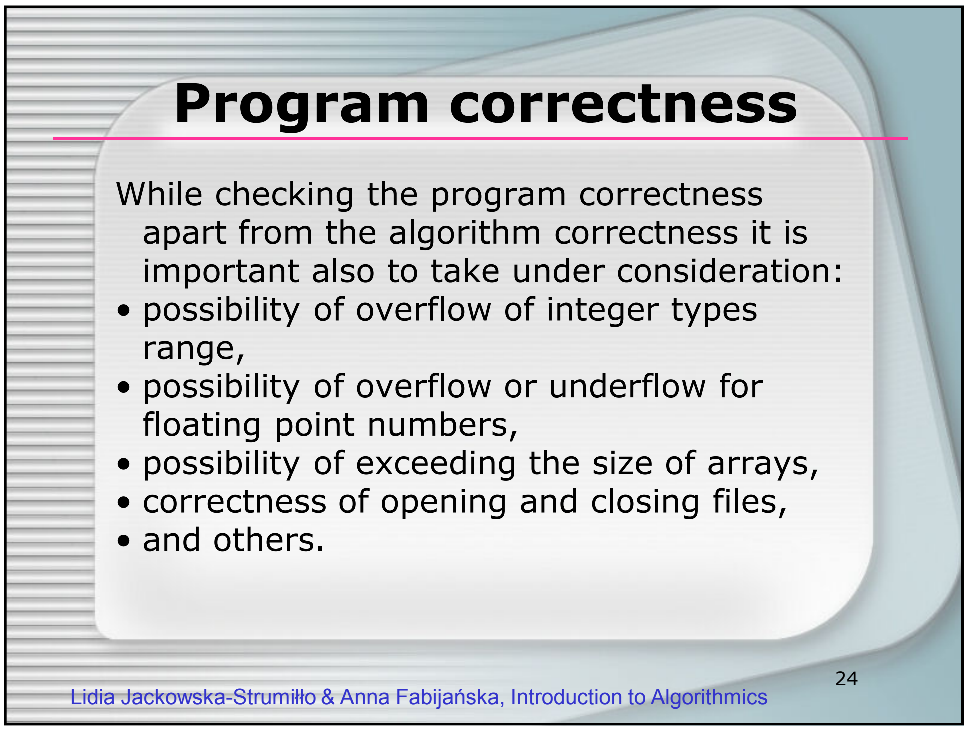
- **loop with a guard:** reads and process data as long as prohibited element occurs
- **loop with a counter:** it is known in advance how many times the loop will be executed
- **general loops:** all others



Halting condition

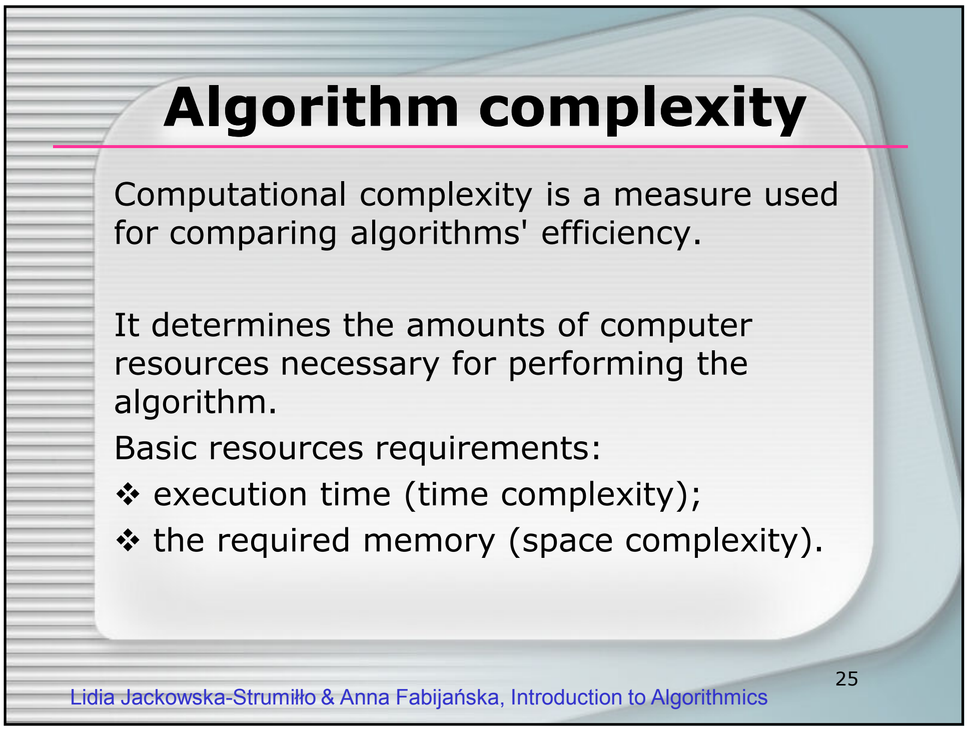
Halting condition determines when the particular program is supposed to terminate.

Program termination is equivalent to:
- correct execution of all the instructions;
- termination of all the loops;
- termination of all the recurrent functions.



Program correctness

While checking the program correctness apart from the algorithm correctness it is important also to take under consideration:
- possibility of overflow of integer types range,
- possibility of overflow or underflow for floating point numbers,
- possibility of exceeding the size of arrays,
- correctness of opening and closing files,
- and others.



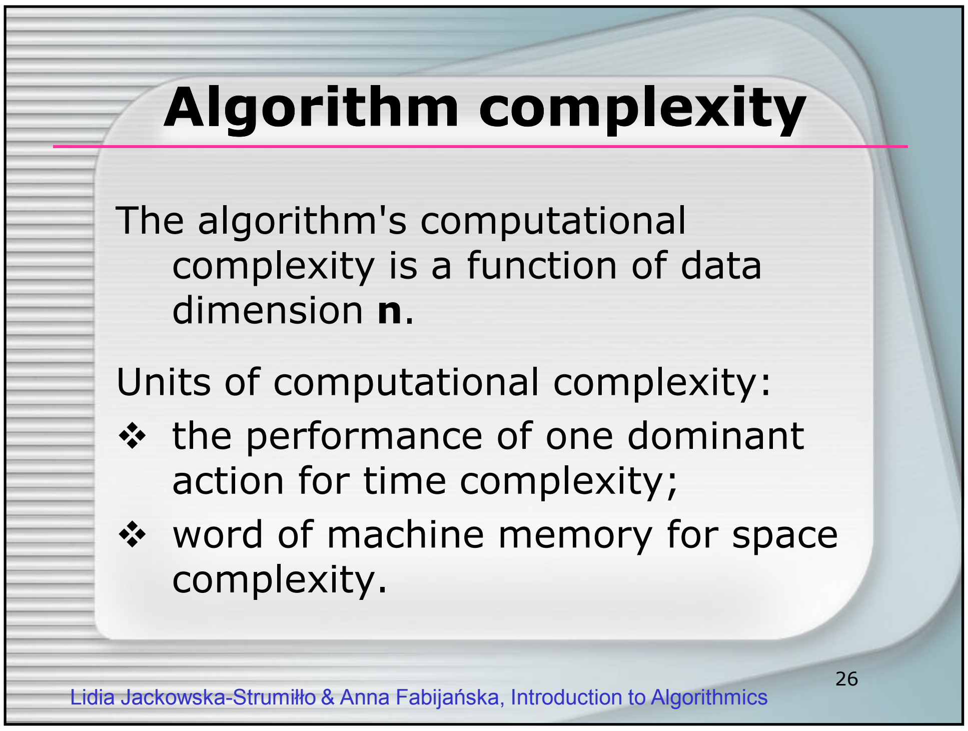
Algorithm complexity

Computational complexity is a measure used for comparing algorithms' efficiency.

It determines the amounts of computer resources necessary for performing the algorithm.

Basic resources requirements:

- ❖ execution time (time complexity);
- ❖ the required memory (space complexity).

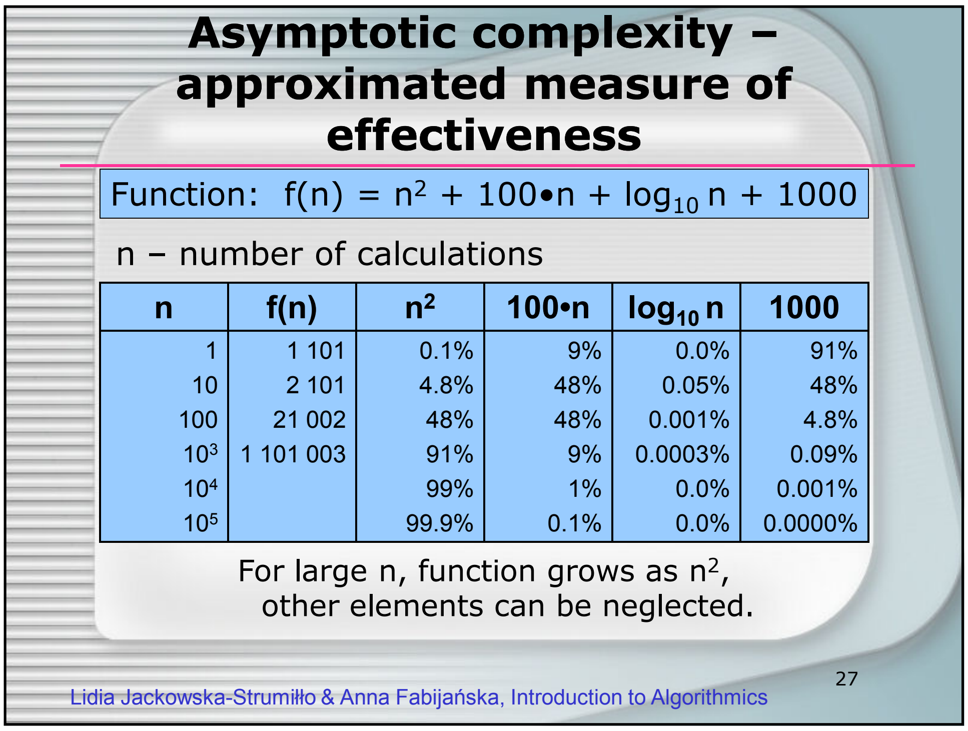


Algorithm complexity

The algorithm's computational complexity is a function of data dimension **n**.

Units of computational complexity:

- ❖ the performance of one dominant action for time complexity;
- ❖ word of machine memory for space complexity.



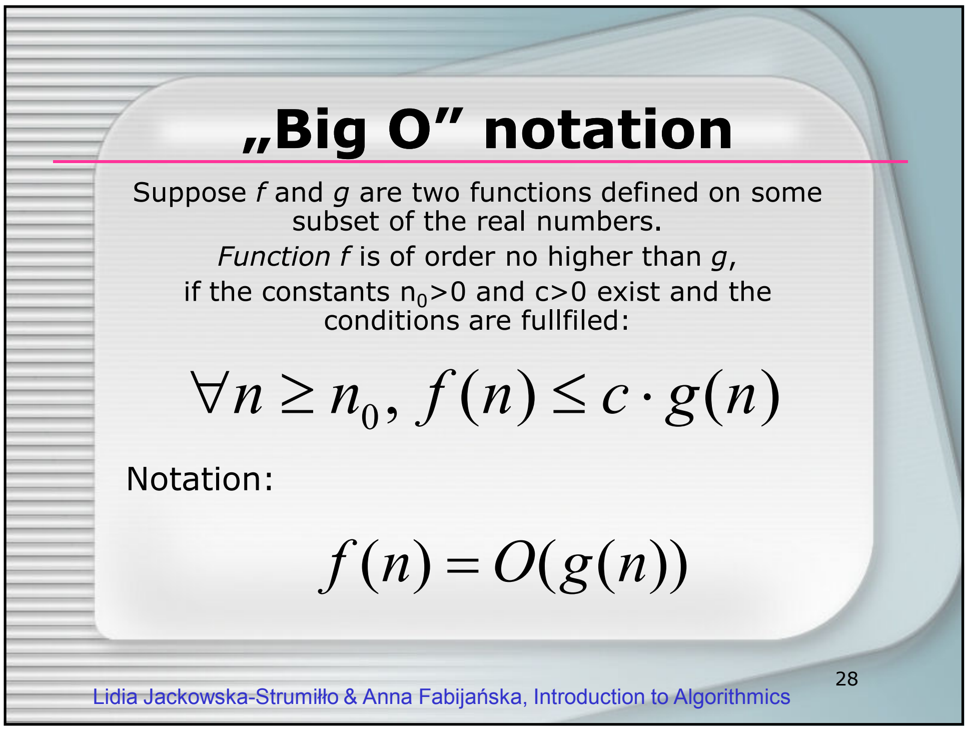
Asymptotic complexity – approximated measure of effectiveness

Function: $f(n) = n^2 + 100 \bullet n + \log_{10} n + 1000$

n – number of calculations

n	f(n)	n^2	$100 \bullet n$	$\log_{10} n$	1000
1	1 101	0.1%	9%	0.0%	91%
10	2 101	4.8%	48%	0.05%	48%
100	21 002	48%	48%	0.001%	4.8%
10^3	1 101 003	91%	9%	0.0003%	0.09%
10^4		99%	1%	0.0%	0.001%
10^5		99.9%	0.1%	0.0%	0.0000%

For large n, function grows as n^2,
other elements can be neglected.



„Big O” notation

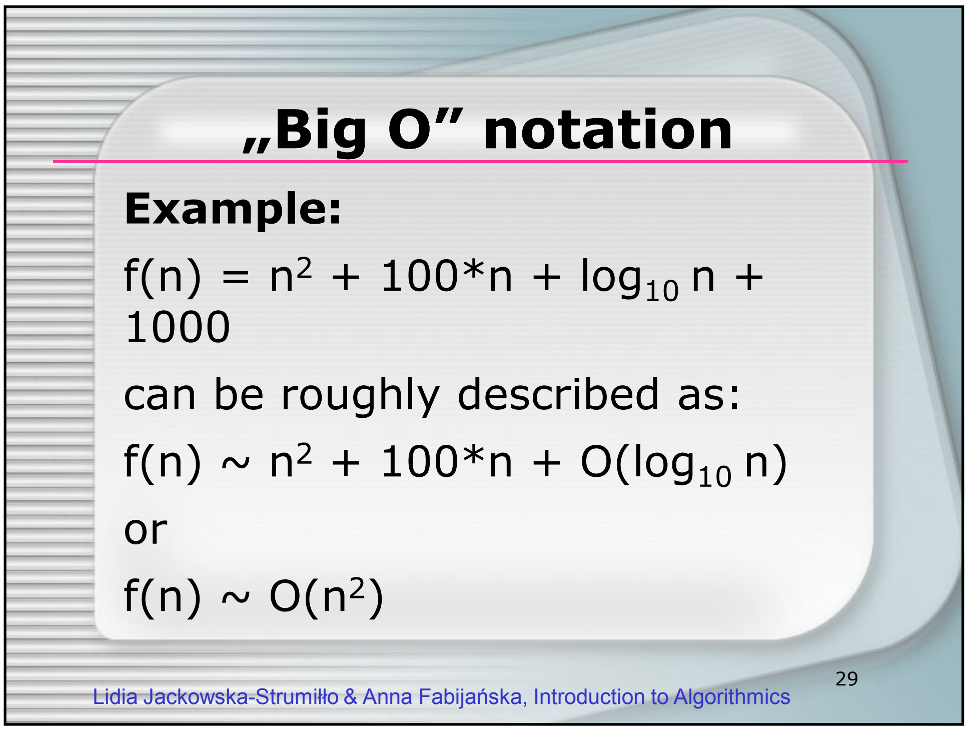
Suppose *f* and *g* are two functions defined on some subset of the real numbers.

Function f is of order no higher than *g*,
if the constants $n_0>0$ and $c>0$ exist and the conditions are fullfiled:

$$\forall n \ge n_0,\ f(n) \le c \cdot g(n)$$

Notation:

$$f(n) = O(g(n))$$



„Big O" notation

Example:

f(n) = $n^2 + 100*n + \log_{10} n +$ 1000

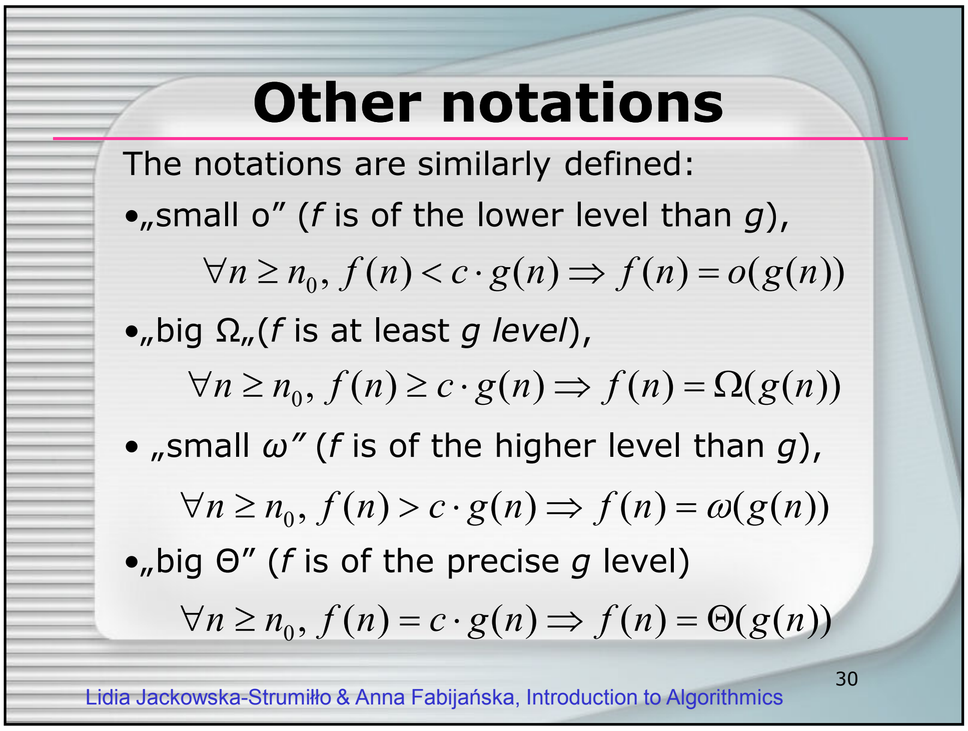
can be roughly described as:

f(n) ~ $n^2 + 100*n + O(\log_{10} n)$

or

f(n) ~ $O(n^2)$

29



Other notations

The notations are similarly defined:

- „small o" (*f* is of the lower level than *g*),

$$\forall n \geq n_0,\ f(n) < c \cdot g(n) \Rightarrow f(n) = o(g(n))$$

- „big Ω„(*f* is at least *g level*),

$$\forall n \geq n_0,\ f(n) \geq c \cdot g(n) \Rightarrow f(n) = \Omega(g(n))$$

- „small *ω"* (*f* is of the higher level than *g*),

$$\forall n \geq n_0,\ f(n) > c \cdot g(n) \Rightarrow f(n) = \omega(g(n))$$

- „big Θ" (*f* is of the precise *g* level)

$$\forall n \geq n_0,\ f(n) = c \cdot g(n) \Rightarrow f(n) = \Theta(g(n))$$

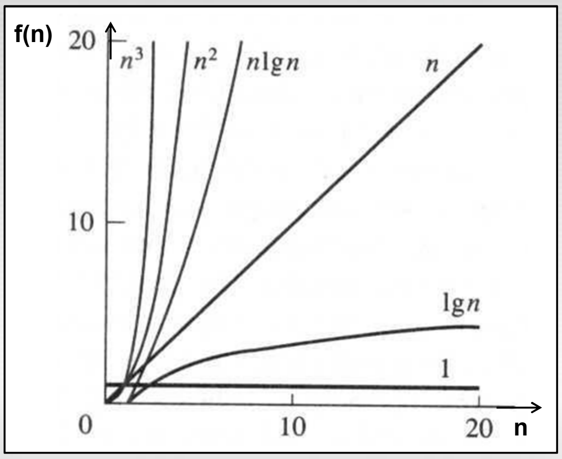
30

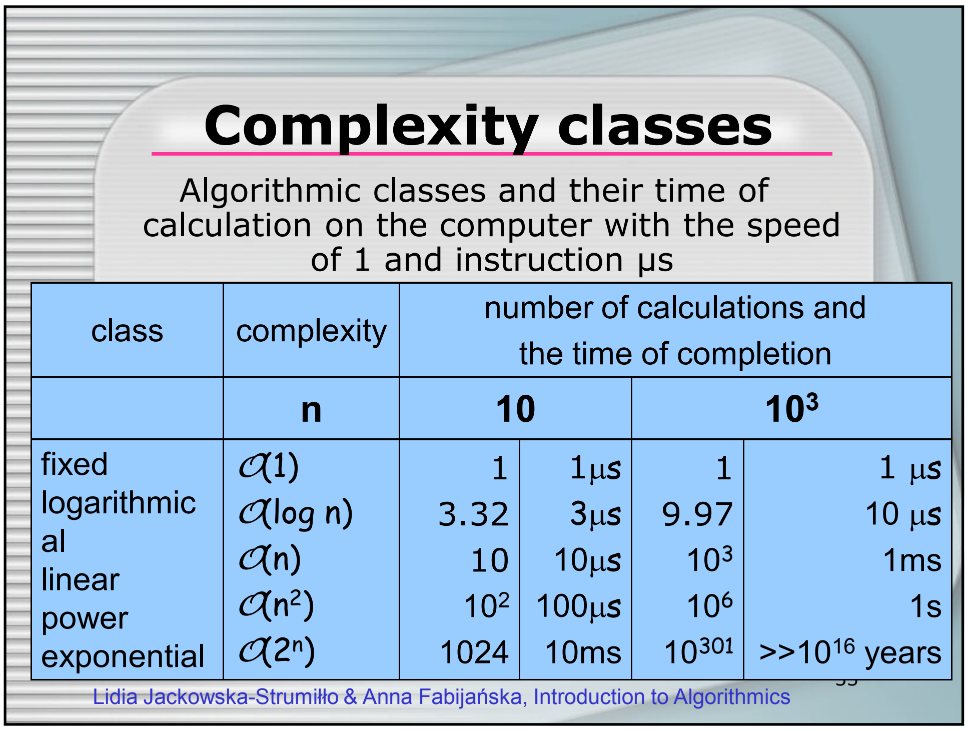
Complexity classes

computational complexity :

- **1** – fixed
- **$\log_2 n$** - logarithmical
- **n** - linear
- **$n \log_2 n$** linearly-logarithmical (or quasi-linear)
- **n^2** - power
- **n^c** – multinominal
- **c^n** - exponential

Complexity classes

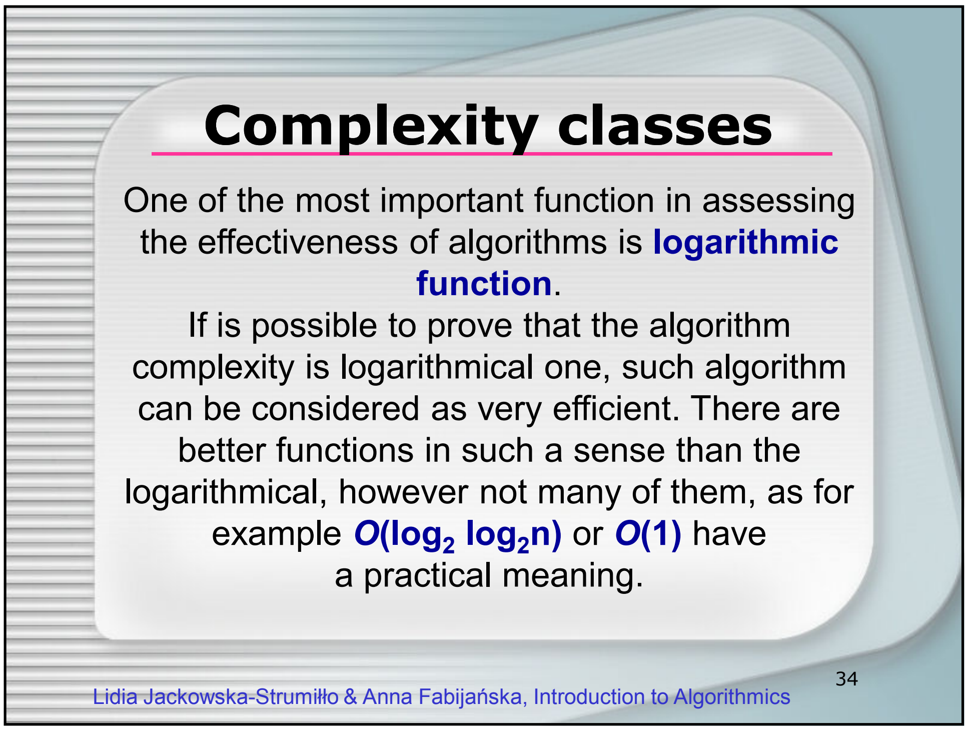




Complexity classes

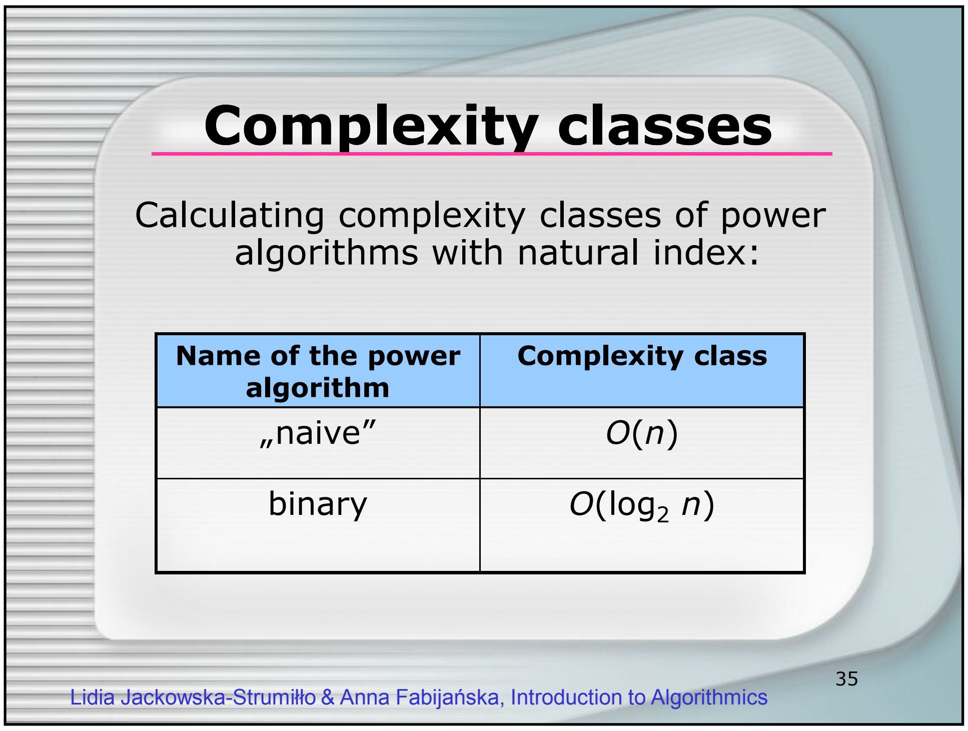
Algorithmic classes and their time of calculation on the computer with the speed of 1 and instruction µs

class	complexity	number of calculations and the time of completion			
	n	10		10^3	
fixed	$\mathcal{O}(1)$	1	1µs	1	1 µs
logarithmical	$\mathcal{O}(\log n)$	3.32	3µs	9.97	10 µs
linear	$\mathcal{O}(n)$	10	10µs	10^3	1ms
power	$\mathcal{O}(n^2)$	10^2	100µs	10^6	1s
exponential	$\mathcal{O}(2^n)$	1024	10ms	10^{301}	$>>10^{16}$ years



Complexity classes

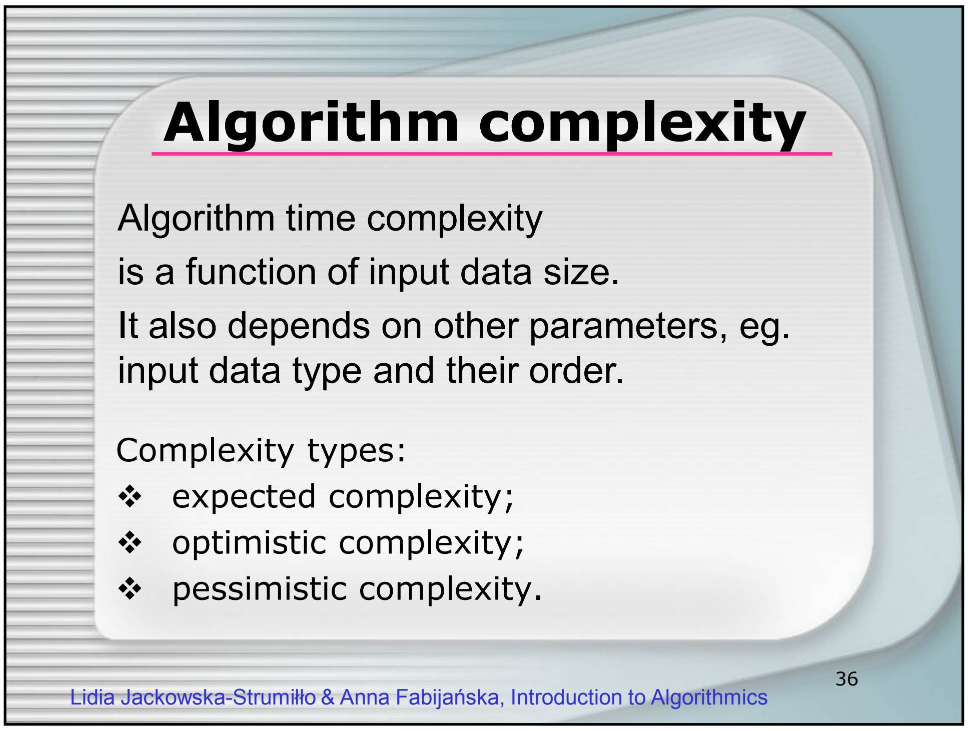
One of the most important function in assessing the effectiveness of algorithms is **logarithmic function**.
If is possible to prove that the algorithm complexity is logarithmical one, such algorithm can be considered as very efficient. There are better functions in such a sense than the logarithmical, however not many of them, as for example $O(\log_2 \log_2 n)$ or $O(1)$ have a practical meaning.



Complexity classes

Calculating complexity classes of power algorithms with natural index:

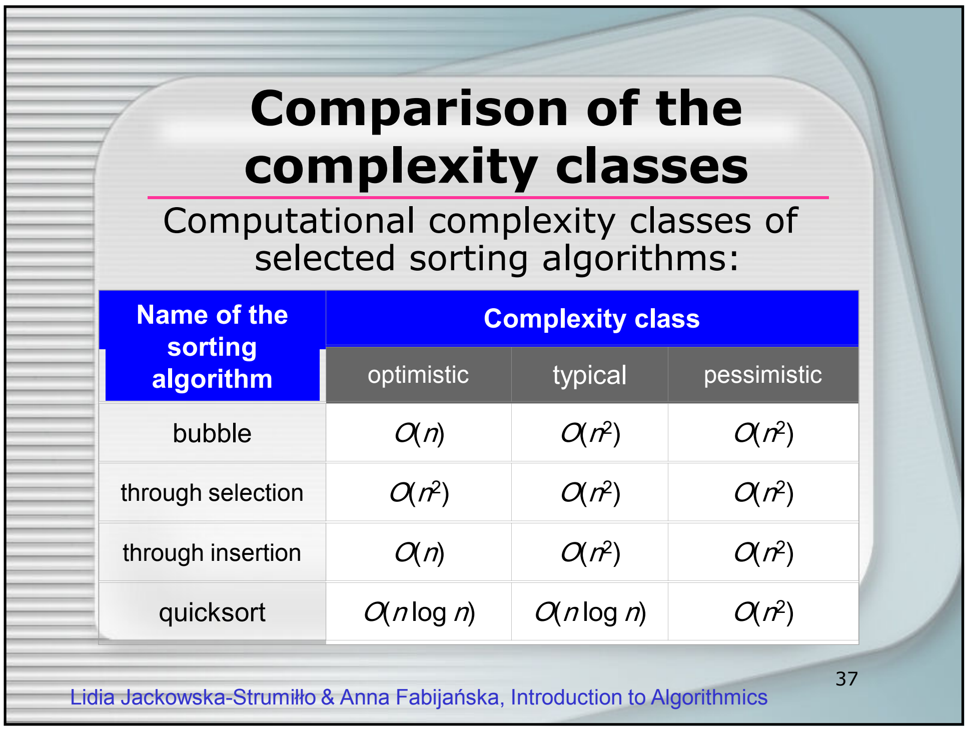
Name of the power algorithm	Complexity class
„naive"	$O(n)$
binary	$O(\log_2 n)$



Algorithm complexity

Algorithm time complexity
is a function of input data size.
It also depends on other parameters, eg. input data type and their order.

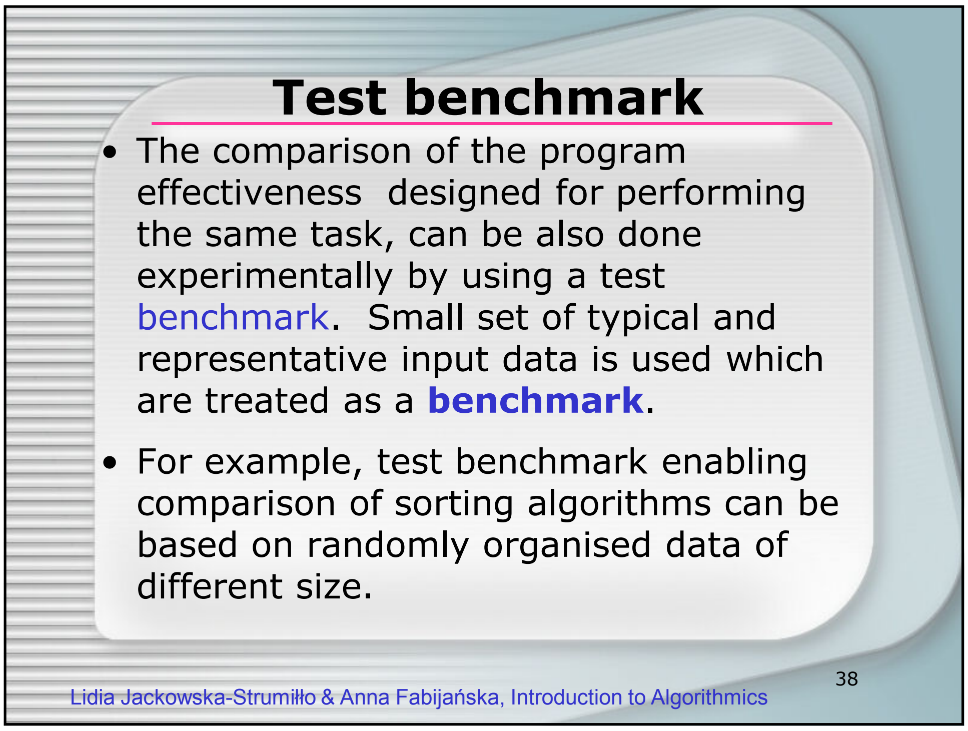
Complexity types:
- ❖ expected complexity;
- ❖ optimistic complexity;
- ❖ pessimistic complexity.



Comparison of the complexity classes

Computational complexity classes of selected sorting algorithms:

Name of the sorting algorithm	Complexity class		
	optimistic	typical	pessimistic
bubble	$O(n)$	$O(n^2)$	$O(n^2)$
through selection	$O(n^2)$	$O(n^2)$	$O(n^2)$
through insertion	$O(n)$	$O(n^2)$	$O(n^2)$
quicksort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$



Test benchmark

- The comparison of the program effectiveness designed for performing the same task, can be also done experimentally by using a test benchmark. Small set of typical and representative input data is used which are treated as a **benchmark**.
- For example, test benchmark enabling comparison of sorting algorithms can be based on randomly organised data of different size.

Time complexity

Execution time is a function **T(n)**, which determines the number of time units, necessary for program or algorithm execution for **n-**size problem.

If the execution time depends on the particular input data (not only their size) the **T(n)** function is defined the worst-case execution time (WCET).

The other measure used for evaluating time complexity is also an *average execution time*, obtained from algorithm runs on different data sets.

Experimental checking of computational complexity

By measuring different execution times of the algorithm for input data sets of different size the algorithm complexity class can be experimentally verified.

Example:

If an algorithm is of time complexity class $O(n^2)$, then the time of algorithm execution for n elements is roughly proportional to the second power of n, therefore:

$$t(n) \approx cn^2$$

where: n - number of processed elements,
$t(n)$ - processing time for n-elements,
c - a constant.

This relationship can be checked experimentally.