Tutorial 9

Exercise 1:

a) Propose and write down in a pseudocode an algorithm that solves the following problem:

INPUT: A natural number n.
QUESTION: Is n a prime?

- Draw a control-flow graph of your algorithm.
- Simulate computations of this algorithm on some inputs (e.g., $n = 0$, $n = 1$, $n = 2$, $n = 3$, $n = 4$, $n = 15$, $n = 16$, etc.). Determine how many steps your algorithm performs for these inputs.

b) Construct and write down in a pseudocode an algorithm solving the following problem (this is a problem of finding prime decomposition of a natural number — also called factorization):

INPUT: A natural number n, where $n > 1$.
OUTPUT: Primes $p_1, p_2, \ldots, p_k$ such that $p_1 \cdot p_2 \cdot \ldots \cdot p_k = n$.

Remark: The algorithm can be created by modification and extension of the algorithm constructed in the previous item, or this algorithm can be used as a subroutine.

Exercise 2: Algorithm 1, given below, should solve the following problem:

INPUT: A natural number n.
OUTPUT: Value $n!$ (the factorial of n).

Recall that the function factorial is defined as follows:

- $0! = 1$,
- $n! = (n-1)! \cdot n$ for $n \geq 1$.

For simplicity, assume that the values of variables can be arbitrarily big natural numbers.

Algorithm 1: Computing factorial

```
FACTORIAL (n):
    x := 1
    for i := 2 to n do
        x := x * i
    return x
```

a) Draw the control-flow graph of this algorithm.

b) Describe a computation performed by this algorithm when it obtains number 5 as an input. Write down the sequence of configurations in this computation.

c) Now the goal is to show that the given algorithm is correct, i.e., that for every input it halts after a finite number of steps and gives a correct output.

In case of Algorithm 1, it is convenient to divide the analysis into two parts — the analysis of the case $n = 0$, and the analysis of those cases where $n \geq 1$.

- Show that the algorithm works correctly for the input where $n = 0$. Here it is sufficient to simulate the computation for this input (and to check that it halts and gives the expected output).

So lets assume that we have an input where $n \geq 1$ (i.e., solve the following items with this additional assumption):

- Formulate hypotheses concerning invariants holding at individual positions in the code (i.e., at nodes of the control-flow graph). Try to propose invariants that can be used to prove the correctness of the algorithm.
- Check that invariants proposed in the previous item really hold.
- Using these invariants, show that if a computation of the algorithm halts, then the algorithm produces the correct output.
- Show that for arbitrary input it holds that the algorithm halts on this input after a finite number of steps.

Exercise 3: Algorithm 2, given below, should be used for finding an element in a sorted array. This is one variant of the algorithm for binary search. In this problem, it is not important what is the exact type of elements of the array. For simplicity we can assume that the elements are integers. The elements of the array are indexed from zero, i.e., if array A has n elements, these elements are denoted $A[0], A[1], \ldots, A[n-1]$.
The algorithm should solve the following problem:

> INPUT: A searched value x, an array A of n elements (where $n \geq 0$), whose elements are sorted from the least one to the greatest one, i.e., for all natural numbers i and j, such that $0 \leq i < j < n$, it holds that $A[i] \leq A[j]$.
>
> OUTPUT: A natural number i giving the index of the first occurrence of value x in array A, or a special value NOTFOUND in the case when value x does not occur in array A.

Remark: For simplicity you can assume that values of variables can be arbitrarily big integers.

a) Draw the control-flow graph of the algorithm.

b) Write down the sequence of configurations in a computation on an input where $x = 6$, $A = [1, 3, 3, 4, 6, 6, 8, 9, 10, 10, 12, 13]$, and $n = 12$.

Algorithm 2: Binary search

```
1  BSearch(x, A, n):
2      ℓ := 0
3      r := n
4      while ℓ < r do
5          k := ⌊(ℓ + r) / 2⌋
6          if A[k] < x then
7              ℓ := k + 1
8          else
9              r := k
10     if ℓ < n and A[ℓ] = x then
11         return ℓ
12     return NotFound
```

c) Let us say that we would do the following changes in the algorithm (always only one of these changes). For each of these changes, find an example of an input, for which the algorithm does not work correctly (e.g., accesses elements of the array at indexes out of bounds, returns an incorrect value, etc.).

   (a) On line 4, change condition $\ell < r$ to $\ell \leq r$.

   (b) On line 7, change assignment $\ell := k + 1$ to $\ell := k$.

   (c) On line 9, change assignment $r := k$ to $r := k - 1$.

   (d) On line 9, change assignment $r := k$ to $r := k + 1$.

   (e) On line 5, change assignment $k := \lfloor (\ell + r) / 2 \rfloor$ to $k := \lceil (\ell + r) / 2 \rceil$ (resp. to $k := \lfloor (\ell + r + 1) / 2 \rfloor$).

d) Propose suitable invariants that according to you hold in nodes of the control-flow graph.

   Hint: Before execution of test $\ell < r$ on line 4, the following should hold:

   - $0 \leq \ell \leq r \leq n$,
   - for each i such that $0 \leq i < \ell$ is $A[i] < x$,
   - for each i such that $r \leq i < n$ is $A[i] \geq x$.

e) Verify that the invarints proposed in the previous item really hold.

f) Find out whether the algorithm halts for every input. If this is the case, prove it, if not, give an example of an input, for which the computation does not halt.

g) Based on the previous analysis, either justify that the algorithm is correct, or give an example of an input, for which it does not behave correctly.

h) Let us say we have an implementation of this algorithm where values of variables n, ℓ, r, and k are represented as 32-bit signed integers (i.e., the numbers whose values can be from interval $-2^{31}, \ldots, 2^{31} - 1$) and also all arithmetic operations with these variables are

performed on this datatype. In this implementation, does the algorithm, in its current form, work correctly for all inputs where $n < 2^{31}$?

Exercise 4: Design an algorithm for the following problem. This is a problem where the goal is to assign colors from some given set of colors to nodes of a graph in such a way that no neighbouring nodes are colored with the same color. Colors are denoted by numbers $1, 2, \ldots, k$, where k is the total number of available colors. If we have a graph $G = (V, E)$, where V is the set of its nodes and E the set of its edges, a ***coloring*** of the graph G is a function $f : V \to \{1, 2, \ldots, k\}$, where for each edge $\{u, v\} \in E$ (where $u, v \in V$) it holds that $f(u) \neq f(v)$.

INPUT: An undirected graph $G = (V, E)$ and a natural number k.

OUTPUT: A coloring of the graph G with k colors, or information that there is no such coloring.

Remarks:

- You can assume that nodes of graph G are denoted by numbers $1, 2, \ldots, n$ (where n is the total number of nodes), and that graph G is represented in an input in a form, where number n is given together with a list of edges where every edge is represented as a pair of numbers specifying the nodes connected by the given edge.
- It can be reasonable to divide the whole solution into several subroutines (functions, procedures, methods, ...) solving individual subproblems. For subroutines that solve simple subproblems where it is clear how to implement them, it is not necessary to describe the given subroutine in detail (e.g., by pseudocode), but it is sufficient to describe informally *what* the subroutine should do (it is not necessary to describe *how* it will do it).

  On the other hand, key parts of the algorithm should be described precisely and in detail, preferably by a pseudocode, is such a way that their implementation in some programming language would require only straightforward rewriting of the given pseudocode to this programming language.
- It can be convenient to use recursion for the solution.
- Informally justify that your algorithm is correct, i.e., give reasons why it holds that the algorithm halts for every input and gives a correct output.