AN INTRODUCTION TO PROGRAM CORRECTNESS AND COMPLEXITY
(DRAFT NOTES)

RAUL H.C. LOPES

1. Algorithms

Algorithms have been frequently characterized as finite descriptions of computations that should always be effective (mechanically executable), terminating (the algorithm comes to an end after a finite time for any input), and deterministic (the course of execution is uniquely defined for a given input.) (See initial chapters of [1] and [11] for considerations on the concept of algorithm.)

Experience with randomized algorithms (see [9]) and with non-determinism as a tool for specification of concurrent and sequential algorithms [4] suggests incorporating non-determinism, parallelism, and non-termination as admissible characteristics of algorithms (Knuth (in [7]) prefers the name *computational method* when non-termination and non-determinism are admitted.)

1.1. **Algorithm characterization.** An algorithm can always be characterized with respect to, at least, the following aspects:

- **Finiteness of description:** every algorithm should have a finite textual representation
- **Effectiveness:** each one of its step should be mechanically executable.
- **Termination:** Does the algorithm come to an end after a finite number of *transitions*?
- **Determination:** is the course of execution for a given input uniquely defined?
- **Efficiency:** How many resources (memory, transitions/time) does the algorithm demand?

1.2. **Non-determinism and concurrency.**

1.2.1. *Pre-algorithmic formulation.* Bauer and Wössner [1] introduce the idea of using *pre-algorithmic* construction in the first stages of problem specification.

Algorithm 1.

$$\boxed{\begin{array}{l}\underline{\textbf{funct}}\ gcd(a \in \underline{\textbf{Nat}} b \in \underline{\textbf{Nat}}) \longrightarrow \underline{\textbf{Nat}} \equiv \\ \{\textbf{Requires}: a \neq 0 \vee b \neq 0\} \\ \langle \eta\ t : t \in \underline{\textbf{Nat}} : t \mid a \wedge t \mid b \wedge \langle \forall\ y : y \in \underline{\textbf{Nat}} : y \mid a \wedge y \mid b \Rightarrow y \leq t \rangle\rangle\end{array}}$$

Definition above states that gcd is a function that when given two arguments in $\underline{\textbf{Nat}}$ (intuitively the set of natural numbers), with the pre-condition that at least one of them is not zero, produces some number t such that:

- $t \in \underline{\textbf{Nat}}$
- t divides (symbol $|$) both a and b;
- for any $y \in \underline{\textbf{Nat}}$ that divides a and b, $y \leq t$.

η is a choice operator: a non-deterministic one that chooses any element from a given set. Gurevich in his *Abstract State Machines* ([6]) also defines a choice operator.

The *function* in 1 does not state a finite sequence of steps to find the greatest common divisor. It is, however, useful as a first definition of what must be produced. Bauer and Wössner refine this definition to 2, where the operator ι is used in the sense of selecting the unique element of a given set that satisfies some property. Should there be more one element satisfying the property and the computation would *diverge*, for example, finishing with undefined (Ω) result.

Algorithm 2.

$$\boxed{\begin{array}{l}\underline{\textbf{funct}}\ gcd(a \in \underline{\textbf{Nat}} b \in \underline{\textbf{Nat}}) \longrightarrow \underline{\textbf{Nat}} \equiv \\ \{\textbf{Requires}: a \neq 0 \vee b \neq 0\} \\ \langle \iota\ t : t \in \underline{\textbf{Nat}} : t \mid a \wedge t \mid b \wedge \langle \forall\ y : y \in \underline{\textbf{Nat}} : y \mid a \wedge y \mid b \Rightarrow y \leq t \rangle\rangle\end{array}}$$

Exercise 1. *Prove that 2 does not diverge.*

1.2.2. *Logical description.* A program to evaluate the maximum of a given set of numbers could be initially described as the function in 3.

Algorithm 3.

$$\boxed{\begin{array}{l}\underline{\textbf{funct}}\ max(S \subset \underline{\textbf{Nat}}) \longrightarrow \underline{\textbf{Nat}} \equiv \\ \langle \eta\ x : x \in S : \langle \forall\ y : y \in S : y \leq x \rangle\rangle\end{array}}$$

Fact: An execution of a program can be represented as the sequence of states.

- **state**: a state of a program is an assignment of values to all its variables.

- **action:** a temporal logic expression, defining a relation between *old states* and *new* states. For example,

$$p \ \mathbf{co} \ q$$

that states (using Hoare triples)

$$\forall \alpha . \{p\} \ \alpha \ \{q$$

if p is true of a given state, after any action of the system in that state, q will hold in the resulting state.

An action is a logical relation constraining the *transitions*, or *updates* in the language of [6]. (I'll use the terms *transition* and *update* to designate synonyms for the usual *step* of a program, and say that actions constraining the transitions/steps of a program.)

A logical description of a program defines a set of conditions that any implementation of it should satisfy. Misra, in [8, 3], uses a formal system (called UNITY), which combines a temporal logic to present logical descriptions of programs and a language (whose semantics is expressed in the logic of UNITY) for representing operational descriptions of programs. As an example, he introduces a program for computing the maximum of a set of numbers as is given in 4. Notice the use of the operator **co** described above.

Algorithm 4.

program max
initially $v = -\infty$
$\alpha : v = m \ \mathbf{co} \ v = m \vee (v \in S \wedge v > m)$
$\beta : m \in S \leadsto v \geq m$

In a logical description, the steps to be performed are described through predicates that represent logical relation between states. In a sense, 4 above describes a family of programs. The programs in that family should obey the following conditions:

- **initially** $v = -\infty$

  Before any computation v must be initialized to ∞.
- $v = m \ \mathbf{co} \ v = m \vee (v \in S \wedge v > m)$

  which states that $\forall m . \forall \alpha : \{v = m\} \ \alpha \ \{v = m \vee (v \in S \wedge v > m)$

  For any *transition* in this program, if v equals m before the transition, then after it either v is still equal to m or it has been changed to some value from S and $v > m$.

- $m \in S \leadsto v \geq m$
  that establishes that for any m if $m \in S$ in a given state, then eventually v will be greater or equal than m.

1.2.3. *States and transitions.*

- **state:** a snapshot of a given execution: the assignments of values to variables;
- **action:** a logical relation between states; it restricts the possible transitions from a state.
- **behaviour (run or execution:** a sequence of state $\langle s_0, s_1, s_2, ldots \rangle$, such that s_0 is produced by the initialization of the program and each s_{i+1} is obtained from s_i by some transition of the given program.
- **stable property:** a property whose truth is not affected by any transition: if it holds before; it still holds after the transition.
- **invariant property:** a property the holds after initialization and is stable.
- **safety conditions:** expressed through invariant properties they establish that the transitions are safe, in Lamport's words, *bad things don't happen.*
- **progress conditions:** they state the program/system is on the move.

1.2.4. *Proving program properties.* Given a program, maybe through a logical description, one is usually interested in investigating at least the following properties of that program:

- *safety* of the results: does it always compute the expected output?
- *termination* of computation: does the computation ever reach a halting state, when no variables are changed and nothing useful is computed?
- *complexity* of computation: how long does it take to reach the halting state?

The following definition will be used in the study of program properties.

Definition 1.

- p **stable** $\equiv p$ **co** p
  A property is stable when its truth is not changed by any transition: if p is true in a state, it will still be true after any transition from that state.

- p **invariant** $\equiv$ <u>**initially**</u> $p \wedge p$ **stable**
  A property about a program is invariant if it is true after initialization and is kept true for any execution of the program.

Notation 1. *The notation of [?] will be followed and curried notation will be used for function application, using*

$$f.i.j$$

to denote the aplication of f to i and j. As usual, an application to a tuple of values is acceptable as in

$$max(x, y)$$

A sequence of τ is understood as a mapping from integers to τ, and in principle an array will be seen a function.

We can prove about 4 that

$$v \leq max.S \textbf{ stable}$$

where

$$max.S = \langle \eta\ x : x \in S : \langle \forall\ y : y \in S : y \leq x \rangle \rangle$$

Proof:

(1) $\forall m : v = m$ **co** $v = m \vee (v \in S \wedge v > m)$
  $\langle$**from** $\alpha\rangle$
(2) $v = m_0$ **co** $v = m_0 \vee (v \in S \wedge v > m_0)$
  $\langle\alpha$ for $m = m_0\rangle\rangle$
(3) $m_0 \leq max.S$
  $\langle$**stability of** $v \leq max.S\rangle$
  (a) either $v = m_0$, and as such $v \leq max.S$
    $\langle$**first outcome of** $\alpha\rangle$
  (b) or $(v \in S \wedge v > m)$, and $v \leq max.S$ by the first conjunct.

□

It can also be proved about the program 4 that *eventually* $v \geq max.S$ also written as

- $\Diamond v \geq max.S$
- $true \leadsto v \geq max.S$

Proof:

(1) $m \in S \leadsto v \geq m$ $\langle$**from** $\beta\rangle$
(2) $max.S \in S \leadsto v \geq max.S$ $\langle$**for** $m = max.S\rangle$
(3) $true \leadsto v \geq max.S$
  $\langle$**from the truth of** max.S $\in$ S$\rangle$

□

Misra, in [3], presents formal derivations for the properties above.

1.2.5. *Operational description.* A possible refinement, still in UNITY and presented in [8], of 4 follows in 5.

Algorithm 5.

<u>**program**</u> *assyncmax*
<u>**declare**</u> $v, i \in \underline{\mathbf{Nat}}$
<u>**initially**</u> $v := -\infty$;
<u>**assign**</u>
$\langle [\!] \; i : 0 \leq i < \natural.S : v =: max(v, S.i) \rangle$

Notice the use of *max* as a binary operator: it is supposed to yield the maximum of its two arguments.

Notation 2.

$$\natural.S$$

will be used to denote the number of elements of S, when S is either a set or a sequence.

In UNITY, the statements in an *assign-section* are separated by the symbol $[\!]$, that indicates an assynchronous execution of the statements in the section, each one being atomic. In this draft, strong fairness is assumed. For example, it is assumed that the program 5 executes

$$v = max(v, S.0), v = max(v, S.1), v = max(v, S.2),$$

forever an in a non-specific order, with each $v = max(v, S.i)$ being chosen (executed) infinitely often.

- this a non-deterministic formulation;
- essentially parallel;
- eventually some i, such that $S.i = max.S$, will be chosen and v will be equal to $max.S$.

1.3. **Fixpoints.**

Definition 2. ***Fixpoint*** $\triangleq$ *a state of the computation where no further work produces any change in the program's variables.*

Program 5 reaches a fixpoint when $m = max.S$.

Proof:

(1) $\mathbf{FP} \equiv \langle \forall\, i : 0 \leq i < \natural.S : v = max(v, S.i) \rangle$
The fixpoint is reached when v doesn't change, and that means v is $max(v, S.i$ for any i.
(2) $\mathbf{FP} \equiv \langle \forall\, i : 0 \leq i < \natural.S : v \geq S.i \rangle$
(3) $\mathbf{FP} \equiv v \geq \langle max\ i : 0 \leq i < \natural.S : S.i \rangle$

$\square$

2. Examples

A set of examples dealing with: invariant properties, loop guards, bound functions, algorithmic complexity.

2.1. Maximum again.

A further, deterministic, refiniment for that program follows in 6.

Algorithm 6.

program *maximum*
declare $v, i \in$ **Nat**
initially $v, i := -\infty, 0;$
assign
$i < \natural.S \longrightarrow v, i := max(v, S.i), i + 1;$

2.1.1. *Fixpoint.*

- $v, i := max(v, S.i), i + 1$ is the equivalent of
  $\|v := max(v, S.i)$
  $\|i := i + 1$
  and that means the statements are executed in parallel and they should be treated as atomic.
- **FP** $\equiv i \geq \natural.S$
- i is incremented in each transition: thus, eventually $i \geq \natural.S$

2.1.2. *Bound function.*

- *maximum* can be seen as scanning the sequence S from left to right.
- A *bound function* (also called *derived variable*) can be defined to establish that *work* must still be performed. It is a function of the variables of the program. It is introduced to define a limit on the transitions that must still be performed: when such limit is reached, no other transition will produce useful changes in the program's variables.
- let t (the bound function) be $\natural.S - i$.
- It can easily be checked that
  (1) assuming that t' is value of t after a transition, it can be proved that, for any transition, either $t > t'$ or $t = t' \wedge t = 0$ (when no transition is performed that changes the programs variables: after $t = 0$, t doesn't change anymore.)
  Notation 3. *Given a transition α and a variable x that can (possibly) be changed by α, x' will be used to denote the value of x after α.*

Proof:

$$
\begin{aligned}
t &= \natural.S - i && \text{definition of } t\\
t' &= \natural.S - i' \\
i' &= i + 1 && \text{program inspection}\\
t' &= \natural.S - (i+1)\\
&= \natural.S - i - 1\\
&= t - 1
\end{aligned}
$$

□

(2) when $i = \natural.S$, $t = 0$.
(3) t gives at any moment the size of task ahead: in this problem, it gives exactly the number of transitions to be performed.
(4) initially the number of tansitions to be performed is $\natural.S$.

2.1.3. *Complexity.* maximum always performs $\natural.S$ transitions when given a set S. Its complexity is said to be $O(\natural.S)$.

Definition 3. $g(n) = O(f(n))$ *means that there are constants* k *and* n_0 *such that for all* $n > n_0$

$$g(n) \leq kf(n)$$

2.1.4. *Invariant property.*

- $P \triangleq (0 \leq i \leq \natural.S) \wedge (v = \langle max\ k : 0 \leq k < i : S.k \rangle)$
- P **stable**

Proof: Assuming P is true before a transition, there are two cases:

(1) $i = \natural.S$, before the transition: the variables don't change and P remains true (fixpoint case.)
(2) $i < \natural.S$ before the transition

$$
\begin{aligned}
&i' = i + 1\\
&i' - 1 = i < \natural.S\\
&i' < \natural.S + 1\\
&i' \leq \natural.(S) && ;\ P'\text{: \textbf{1st conjunct}}\\
&v' = max(v, S.i)\\
&v' = max(\langle max\ k : 0 \leq k < i : S.k \rangle), S.i) && ;\ \text{from } P\\
&v' = \langle max\ k : 0 \leq k \leq i : S.k \rangle && ;\ P'\text{: 2nd conjunct}
\end{aligned}
$$

□

- P **invariant**: from P **stable** and from the truth (trivially checked) of P after initialization.
- When fixpoint is reached, P holds, and consequently v holds the maximum of all $S.i$.

2.2. A multiplication algorithm.

Algorithm 7.

$$\begin{array}{|l|}\hline
\underline{\textbf{program}}\ mult \\
\{\textbf{Requires}: A, B \in \underline{\textbf{Nat}}\} \\
\{\textbf{Ensures}: z = A * B\} \\
\underline{\textbf{declare}}\ x, y, z \in \underline{\textbf{Nat}}; \\
\underline{\textbf{initially}}\ x, y, z := A, B, 0; \\
\underline{\textbf{assign}}\ y > 0 \longrightarrow [\!]\, even(y) \longrightarrow x, y := 2 * x, y/2 \\
\qquad\qquad\qquad\quad [\!]\, odd(y) \longrightarrow y, z := y - 1, z + x \\ \hline
\end{array}$$

- let $P \stackrel{\triangle}{=} x * y + z = A * B$
- **initially** P
  Because initially $x = A, y = B, z = 0$.
- P **stable**

  Proof: Assuming that P holds before the transition.
  - $y = 0$ then there are no changes in variables and P holds after the transition.
  - $y > 0$ then $P' \equiv x' * y' + z' = A * B$

when y is odd:: $x' = x, y' = y - 1, z' = z + x$

$$\begin{aligned}
x' * y' + z' &= x * (y - 1) + (z + x) \\
&= xy - x + z + x \\
&= x * y + z \\
&= A * B \text{ ; because } P \text{ holds before}
\end{aligned}$$

when y is even: $x' = 2 * x, y' = y/2, z' = z$

$$\begin{aligned}
x' * y' + z' &= (2 * x) * (y/2) + z \\
&= xy + z \\
&= A * B \text{ ; because } P \text{ holds before}
\end{aligned}$$

Thus P holds after the transition.

□

- P **invariant**
  Because **initially** P and P **stable**.

2.2.1. *A fixpoint.* The algorithm eventually reaches a fixpoint.

Proof:

- **initially** $y \geq 0$ because $B \in$ **Nat**
- every transition (while $y > 0$) either decrements y by 1 or divides y by 2
- Thus, eventually, y becomes 0 and no other changes are made in the program's variables.

$\square$

When the fixpoint is reached $y = 0$ and the equality $x * y + z = A * B$ establishes

$$z = A * B$$

2.3. **The complexity.** The algorithm takes $O(\lg B)$ transitions to reach a fixpoint.

- We assume B is a power of 2 and define

$$T(0) = 0;\ \text{when } y = 0 \text{ the fixpoint has been reached} \tag{1}$$

$$T(1) = 1 \tag{2}$$

$$T(2^k) = 1 + T(2^{k-1});\ \text{for } k > 0 \text{ each transition divides } y \text{ by } 2 \tag{3}$$

- $T(y)$ counts the number of transitions that must be performed to reach a fixpoint, for a given y. Given a $y > 0$ the number of transitions to be be performed is 1 (the one that divides y by 2) plus the number of transitions necessary for $y/2$ (given by $T(y/2)$.)
- $T(y)$ above is a *recurrence relation* and it is not hard to see and prove that $(1 + \lg y)$ is a solution for that relation.
- Equation 3 above can be solved by a method called iterationwhich consists in recurrently replacing the right-hand side occurrence of T by its definition:

$$\begin{aligned}
T(2^k) &= 1 + T(2^{k-1}) \\
&= 1 + (1 + T(2^{k-2})) \\
&= 1 + (1 + (1 + T(2^{k-3}))) \\
&= \underbrace{1 + 1 + \ldots + 1}_{k} + T(1) \\
&= k + 1
\end{aligned}$$

- That $T(n) = (1 + \lg n)$ is proved by induction:

  Proof:

  when n=1: $(1 + \lg n) = 1 = T(1)$

  assuming it true for $n < 2^k$

  $$\textbf{for } n = 2^k\textbf{:} \qquad \begin{aligned} T(2^k) &= 1 + T(2^k) \\ &= 1 + (1 + \lg 2^{k-1} \\ &= 1 + k \end{aligned}$$

  □

- when y is odd, we add to the recurrence relation
  $$T(2 * n + 1) = 1 + T(2 * n), \text{ when } y \text{ is odd}$$
- It can be proved by induction that if there is m such that
  $$2^m < (2n + 1) < 2^{m+1}$$
  then
  $$T(2n + 1) < T(2^{m+1})$$
- From that, we conclude that $T(n) = O(\lg n)$.

Exercise 2. *Prove that $T(n) = O(\lg n)$ when $T(n)$ is defined as above.*

Note 1. *Program 7 is adapted from* [5].

2.4. **Linear search.** Given: a sequence S with n integers and an integer x. We are asked to find the first occurrence of x in the sequence.

Algorithm 8.

$$\begin{array}{|l|}\hline
\underline{\textbf{program}}\ linearSearch \\
\{\textbf{Requires} : S \in \{0 \ldots n-1\} \longrightarrow \underline{\textbf{Int}}, x \in \underline{\textbf{Int}}, x \in S\} \\
\{\textbf{Ensures} : (i = n \wedge x \notin S) \vee (0 \leq i < n \wedge x = S.i)\} \\
\underline{\textbf{declare}}\ i \in \underline{\textbf{Nat}}; \\
\underline{\textbf{initially}}\ i := 0; \\
\underline{\textbf{assign}}\ i < n \longrightarrow [\!] x \neq S.i \longrightarrow i := i + 1 \\
\qquad\qquad\qquad\quad [\!] x = S.i \longrightarrow skip \\ \hline
\end{array}$$

Notation 4. *skip does nothing.*

2.4.1. *Invariant.* Define
$$P \triangleq (0 \leq i \leq n) \wedge (\langle \forall\ k : 0 \leq k < i : x \neq S.k\rangle$$

Exercise 3. *Prove*
$$P \textbf{ invariant}$$

2.4.2. *Fixpoint.* The program reaches a fixpoint when $i = n$ or $x = S.i$.

2.4.3. *Complexity.* Assuming x occurs in S.

- **Best case:** x is in S and is the first item: one comparison is made and the time spent O(1).
- **Worst case:** $x = S.(n-1)$ and n transitions (comparisons) are performed until the firxpoint is reached.
- **Average case:**
  (1) assign probability p_i to kind of problem
  (2) $\sum_{1 \le i \le n} p_i = 1$
  (3) $T_{\text{average}} = \sum_i p_i T_i$
  (4) In this case, there are n problems: the possible occurrences of x in S
  (5) $p_i = 1/n$
  (6)

$$T = \sum_{1 \le i \le n} i/p_i = \frac{n+1}{2}$$

Exercise 4. *What if we had not assumed that x occurs in S?*

Note 2. *Program 8 is adapted from* [5]*. The analysis of its complexity is presented in* [10].

Exercise 5. *Write a program that, given a sequence $S.(0..n-1)$, will set the following output condition:*

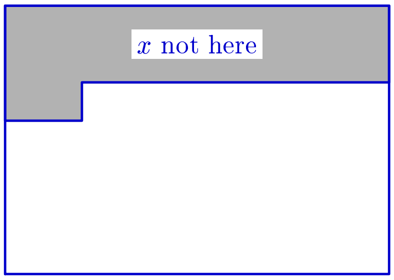
$$x \le S.(0..n-1) \wedge \langle \exists\ j : 0 \le j < n : x = S.j \rangle$$

(1) *Describe your invariant, and bound function.*
(2) *Prove correctness.*
(3) *Consider at least two alternatives (i.e., different programs) representing different search strategies over your sequence.*

(Adapted from [5]***.)***

2.5. **Bidimensional search.** Given a two-dimensional sequence of integers $S.(0..m-1)..(0..n-1)$, and an integer x, find out whether x is in S.

In the figure that follows the box represents the sequence to be searched, and the shaded area, an area that has already been scanned and that is known not to contain x.



The work to be done is to search the whole sequence until x is found. Assume that i and j are used to index the sequence, the target output becomes:

$$R \triangleq (i = m \land x \notin S) \lor (0 \leq i < m \land 0 \leq j < n \land x = S.i.j)$$

From the figure above, an invariant becomes evident: while the search proceeds x *is not the shaded area.*

$$\begin{aligned} P \triangleq 0 \leq & i \leq m \land 0 \leq j < n \\ & \land x \notin S.(0..i-1).(0..m-1) \\ & \land x \notin S.i.(0..j-1) \end{aligned}$$

P expresses the assurance that x has not been found before $S.i.j$, which is at any moment the only reason to proceed with the search.

That defines a safety condition. However, that's not enough: progress must be made and that means extending the shaded area. The unshaded area represents the work to be done:

$$w = (m - i) * n - j$$

Progress is obtained if each step enforces

$$w > w'$$

and that means increasing either i or j. But, increasing i or j means bringing more area into the shaded part of the box, the one that's considered as already scanned and not containing x. Thus,

Restriction: w can be decreased (and the shaded area extended) only if x has not been found: $x \neq S.i.j$.

In addition, there must be assurances that the search will go beyond the limits of S:

Restricton: proceed at most until the whole area is shaded: $w = 0$.

A first refinment of our program follows:

Algorithm 9.

<u>**program**</u> *bidimen*
<u>**initially**</u>
$i, j := 0, 0;$
<u>**assign**</u>
$(m - i) * n - j > 0$ **cand** $x \neq S.i.j \rightarrow$ "increase either i or j"

Increasing either i or j decreases the unscanned area, but that must be orderly, expressing the idea of testing each element of the sequence. Following the figure, j will be increased until n while i is kept fixed and (when j reaches n) i will be increased to start a new line. Finally, when i gets to m w equals to 0: the condition $w > 0$ can be simplified to $i < m$. Program 9 is now refined to:

Algorithm 10.

<u>**program**</u> *bidimen*
<u>**initially**</u>
$i, j := 0, 0;$
<u>**assign**</u>
$i < m$ **cand** $x \neq S.i.j \rightarrow i, j := \begin{cases} i, j + 1, & if\ j < n \\ i + 1, 0, & if\ j = n \end{cases}$

Exercise 6. *Prove that:*

1. *That P is and invariant for program 10*
2. *That $w > w'$ is also an invariant for that program.*
3. *Program 10 has a fixed point.*

Program 10 is developed by Gries in [5] where he develops the loop guard

$$B \triangleq i \neq m \textbf{ cand } S.i.j$$

by observing that B such that when a fixpoint is reached ($\neg B$ is true) P and R must be true. In other words:

$$P \wedge \neg B \Rightarrow R$$

Exercise 7. *Write a program to find the greatest common divisor for two given integers $X > 0$ and $Y > 0$.*

1. *Use only additions and subtractions.*
2. *Prove correctness.*
   - *Use invariant*

$$x > 0 \wedge y > 0 \wedge gcd(x, y) = gcd(X, Y)$$

where x and y are initialized to X and Y.

- *use bound function $w = x + y$. (This time the bound function won't provide the complexity for your algorithm. It will just give the limit when a fixpoint is reached.)*
- *Build your program to ensure that $x > 0$ and $y > 0$ are always true and use that to prove termination.*

2.6. **Bentley's split.** Given is a sequence $S.(0..n-1)$ of integers, and the index p of one of them, called *splitter*.

Target: partition the sequence in 3 subsequences respectivelly composed of the elements less than the splitter, those equal to it, and those greater than it.

An algoritm is presented in [**?**] which consists of 2 steps:

(1) partition the sequence in 4 subsequences, as shown in the figure below:

=	<	>	=

(2) move the 2 subsequences containing elements equal to the splitter to the center, as shown in the figure below:

<	=	>

2.6.1. *Partitioning into 4 subsequences.* Four pointers will be defined to marked the limits of the patitions being constructed:

- **nel**: the insertion point for the next element equal to the splitter on the left side;
- **lt**: the next insertion point for elements less than the spilter;
- **ner**: the insertion point for the next element equal to the splitter on the right side;
- **gt**: the next insertion point for elements greater than the spilter;

The process should start with the pointers marking 4 empty partitions ($nel = lt = 0, ner = gt = n-1$), as illustrated below:

Assuming that $S_0.(0..n-1)$ denotes the initial subsequence, the desired output is given by

$$
\begin{aligned}
&S \text{ is a permutation of } S_0\\
&\wedge\langle\forall\ j : 0 \leq j < nel : S.j = S_0.p\rangle\\
&\wedge\langle\forall\ j : nel \leq j < lt : S.j < S_0.p\rangle\\
&\wedge\langle\forall\ j : gt < j \leq ner : S.j > S_0.p\rangle\\
&\wedge\langle\forall\ j : ner < j \leq 0 : S.j = S_0.p\rangle\\
&\wedge gt < lt
\end{aligned}
$$

Exercise 8. *Define a logical expression for*

$$S \text{ is a permutation of } S_0$$

The four partitions must be initially empty and located at the extremes of the array, corresponding to the following initialization:

$$lt = nel = 0$$
$$gt = ner = n - 1$$

Progress will be achieved by incorporating new elements to any of four partitions being built. At any time, lt will point the first non-incorporated element from the left; gt will point the first non-incorporated element from the right. Each incorporation of some element to one of the four partitions should keep the following true:

$$\begin{aligned} P \triangleq\; & S \text{ is a permutation of } S_0 \\ & \wedge \langle \forall\ j : 0 \leq j < nel : S.j = S_0.p \rangle \\ & \wedge \langle \forall\ j : nel \leq j < lt : S.j < S_0.p \rangle \\ & \wedge \langle \forall\ j : gt < j \leq ner : S.j > S_0.p \rangle \\ & \wedge \langle \forall\ j : ner < j \leq n-1 : S.j = S_0.p \rangle \\ & \wedge \langle \forall\ j : lt \leq j \leq gt : S.j = S_0.j \rangle \end{aligned}$$

A *limit on the progress* will be defined by observing that elements are incorporated from the left by increasing lt and from the right by decreasing gt. As soon as they cross each other, the process should be stopped: all elements having been incorporated into one of the partitions.

The following conditions are thus defined:

- **bounds for the pointers**

$$B_0 \triangleq 0 \leq nel \leq lt$$
$$B_1 \triangleq gt \leq ner < n$$

- partition **safety** is guaranteed by P
- **bound function:**

$$w \triangleq gt - lt + 1$$

w gives the work that remains to be done and

$$w > 0$$

When w reaches 0 no further work should be done.

A program must be written that maintains the following properties:

- P **invariant**, that ensures that the partition process is under control;
- $\Diamond w = 0$, that ensures that work is being performed by each transition and that eventually no work will be left to be done.

In addition, the program must ensure that the bound $w > 0$ is respected: progress doesn't get out of control. When $w = 0$ no more work is performed: a fixpoint has been reached.

The algorithm representing the partitioning of an input vector $S.(0..n-1)$ into 4 partitions is depicted in program 11. It assumes that p contains the initial index for the splitter. It deals with five conditions on input:

(1) the element pointed to by lt is equal to the splitter: it must be moved to the right, and this is done through a swap with the element point to by nel;
(2) the element pointed to by lt is less than the splitter: lt is advanced;
(3) lt and gt point to elements that are respectively greater and less than the splitter: they must be swapped;
(4) the analogous for an element equal to the splitter on the right side;
(5) gt points to an element greater than the splitter: the pointer moves to the left.

Algorithm 11.

```
program split
initially
  sp, nel, lt, ner, gt := S.p, 0, 0, n − 1, n − 1;
assign
  gt − lt + 1 > 0 → if S.lt = sp → swap(S, nel, lt);
                    [] S.lt < sp → lt := lt + 1;
                    [] S.lt > sp ∧ S.gt < sp → swap(S, lt, gt);
                    [] S.gt = sp → swap(S, ner, gt);
                    [] S.gt > sp → gt := gt − 1;
                    fi
```

In the algorithm 11, imagine $swap(S, u, v)$ as an abbreviation (a macro call) for a code defined as

$$swap(S, u, v) \triangleq S.u, S.v, u, v := S.v, S.u, u + 1, v + 1$$

Exercise 9. *Prove that:*

(1) *P as defined above is an invariant for program 11.*
(2) *Given w as above, $\Diamond w = 0$.*
(3) *Program 11 has a fixpoint.*
(4) *When the fixpoint is reached, the output condition R (defined above) is verified.*

Exercise 10. *Extend program 11 to move the two partitions with elements equal to the splitter to center of the vector.*

Exercise 11. *Write a program that, given an array $S.(0..n-1)$ of integers, such that the section $S.(0..m-1)$, for $m < n$, is ordered, orders the section $S.(0..m)$.*

Exercise 12. *Extend the program above to order an array $S.(0..n-1)$.*

Exercise 13. *Write a program to find $k-th$ smallest number of a vector $S.(0..n-1)$ of integers.*

Exercise 14. *Write a program that, given a sequence $S.(0..n-1)$ of integers, sets a variable x to the sum of the maximum consecutive subsequence.*

(1) *State clearly your output condition.*
(2) *Define invariant and bound function.*
(3) *Prove the correctness of your program.*

References

[1] F.L. Bauer and H.Wössner, *Algorithmic lamguage and program development*, Springer-Verlag, 1982.
[2] Manfred Broy (ed.), *(th summer school on constructive methods in computer science*, Springer-Verlag, 1988.
[3] K. Many Chandy and Jayadev Misra, *Parallel program design: A foundation*, Addison-Wesley, 1988.
[4] E.W. Dijkstra, *A discipline of programming*, Prentice Hall, Englewood Cliffs, 1976.
[5] David Gries, *The science of programming*, Springer-Verlag, New York, 1981.
[6] Yuri Gurevich, *Evolving algebras 1993: Lipari guide*, Specification and Validation Methods (E. Börger, ed.), Oxford University Press, 1993.
[7] Donald E. Knuth, *The art of computer programmimg, volume 1: Fundamental algorithms*, Addison-Wesley, 1998.
[8] Jayadev Misra, *A foundation of parallel progragramming*, In Broy [2], pp. 397–433.
[9] Rajeev Motwani and Prabhakar Raghavan, *Randomized algorithms*, Cambridge University Press, 1995.
[10] Paul Walton Purdom, Jr. and Cynthia A. Brown, *The analysis of algorithms*, Holt,Rienhart and Winston, 1985.
[11] Hartley Rogers, *Theory of recursive functions*, MIT Press, 1967.