CMPUT 272 Required reading

# Variants and Invariants

December 2010

Almost every algorithm repeats (iterates, loops) a certain transformation on a state of "things"; in other words, some task is repeatedly performed. There is a defined starting, initial state and an iterative algorithm defines a sequence of states. The next state in the sequence is obtained by a transformation of a previous state according to the algorithm. Such iterative processes are typically programmed using loops or using recursion.

When studying iterative algorithms we look at the following issues (cf. [4]):

- Does the algorithm stop?
- Can a desired state be reached?
- Does the algorithm reach (converge to) a state which does not change?
- What are the reachable states?

The principal strategy for answering such questions consists of finding what changes (a variant) when a transformation is performed and what stays the same (an invariant).

VARIANT is a non-negative integer expression whose value decreases with each loop execution. Variants are used to demonstrate the termination of an iterative process.

INVARIANT is a relationship among elements of the state of an iterative process which holds on as long as the process is executed. Invariants are used for proving what the process does.

Finding useful variants and invariants is a skill which is best learned by examples and we proceed to study several of them. Most of the material presented here is folklore but we try to give references to available sources.

## 1 The Coffee Can Problem

**Problem** [5, p. 165, p. 304]

> A coffee can contains some black beans and white beans. The following process is repeated as long as possible.
>
> Randomly select two beans from the can. If they have the same color, throw them out, but put another black bean in. (Enough extra black beans are available to do this.) If they are different colors, put the white one back into the can and throw the black one away.
>
> Prove that the process stops with exactly one bean in the can. Specify the color of the final bean in terms of the number of white beans and the number of black beans initially in the can.

**Exercise** The above is an informal specification of a process. Are you sure how the process is run?

The following program describes the behavior of the robot manipulating coffee beans in the can. The robot is able to obey some control operations and execute the following operations: *PickaBean*, *PutBlack* and *PutWhite*, whose meaning should be obvious. Putting a bean into the can is always possible. The robot gets stuck when it cannot pick a bean from the can. Is it clear that the robot finishes with one picked bean?

*The coffee can*

```
$bn1, bn2 \in Bean$;
repeat
    $bn1 := PickaBean$; $bn2 := PickaBean$;
    if $bn1.col = white$ & $bn2.col = white$ then
        $PutBlack$;
    elseif $bn1.col = black$ & $bn2.col = black$ then
        $PutBlack$;
    else
        $PutWhite$;
    endif
end
```

*End of the coffee can*

Note that *PickaBean* is non-deterministic: when executed twice with the same can contents it may give different results.

Before proceeding, we make some general remarks. Whenever we design a program we should know what the program is supposed to do. The statement of what the program is supposed to do is called the program **specification**. Each program specification must define the following.

- MATERIAL that is manipulated (input and output data).
- PRECONDITIONS, which is the statement of the restrictions on the input data.

- POSTCONDITIONS, which is the statement of the required properties of the output data.

With the above in mind, we introduce some additional notation into the program in order to be able to talk about the process.

$W$ and $B$ are the initial numbers of white and black beans, respectively.

$w$ and $b$ are the current numbers of white and black beans, respectively, when the process is run.

We state the starting conditions (pre-conditions) for the process by assuming that there are some white beans in the can and there are some black beans in the can.

Using $W$, $B$, $w$ and $b$ we characterize the final state of the process, we call these post-conditions.

*The coffee can*

```
$bn1, bn2 \in Bean$;
$W, B \in \mathbb{N}$
    PRECONDITIONS: $0 < W$ & $0 < B$;
$w, b \in \mathbb{N}$
$w := W; b := B$;
while $1 < w + b$ do
    $bn1 := PickaBean$; $bn2 := PickaBean$;
    if $bn1.col = white$ & $bn2.col = white$ then
        $PutBlack$;
        $w, b := w - 2, b + 1$
    elseif $bn1.col = black$ & $bn2.col = black$ then
        $PutBlack$;
        $b := b - 1$
    else
        $PutWhite$;
        $b := b - 1$;
    endif
endwhile
POSTCONDITIONS
    R1: How many beans are in the can? Answer: $w + b = 1$.
    R2: What are their colors? Answer: $w = W \pmod 2$.
```

*End of the coffee can*

The preconditions say that at the start we have at least one white bean and at least one black bean. In the postcondition *R2* we use the relation of equality modulo defined as follows, see also [1, p. 58].

**Definition.** Let $a$, $b$ and $c$ be natural numbers, $c \neq 0$. $a = b \pmod c$ means that there exists an integer $z$ such that $a - b = c \cdot z$.

When $a = b \pmod 2$ we say that $a$ and $b$ have the same parity, i.e. both $a$ and $b$ are even, or both are odd.

How is the postcondition *R1* answering the question about the colors of the remaining beans? From *R1* we know that there is only one bean left. Then *R2* tells us that the remaining bean is white when the initial number of white beans was odd.

We will now argue that the program is correct, that is, when the program is run in a state satisfying the preconditions, it will terminate in a state satisfying the postconditions.

We have to prove a number of properties of our program. At the heart of the argument is reasoning about the **while**-loop. We will characterize the behavior of the loop in terms of a loop variant and a loop invariant, which we first state.

*The coffee can*

```
$bn1, bn2 \in Bean$;
$W, B \in \mathbb{N}$
    PRECONDITIONS: $0 < W$ & $0 < B$;
$w, b \in \mathbb{N}$
$w := W; b := B$;
while $1 \leq w + b$ do
            VARIANT $w + b$
            INVARIANTS
                  $I1 : 1 \leq w + b$
                  $I2 : w = W \pmod 2$
      $bn1 := PickaBean$; $bn2 := PickaBean$;
      if $bn1.col = white$ & $bn2.col = white$ then
            $PutBlack$;
            $w, b := w - 2, b + 1$
      elseif $bn1.col = black$ & $bn2.col = black$ then
            $PutBlack$;
            $b := b - 1$
      else
            $PutWhite$;
            $b := b - 1$;
      endif
endwhile
POSTCONDITIONS
      R1: How many beans are in the can? Answer: $w + b = 1$.
      R2: What are their colors? Answer: $w = W \pmod 2$.
```

*End of the coffee can*

Why did we choose this particular expression and formulae? Using the variant expression we will argue that the loop terminates. Using the invariant formulas we will prove that the postconditions are met when the loop terminates (and because of the variant we know that it does).

We start with proving the invariant formulas. We have to prove that no matter how many times the loop has been executed the invariant formulas are always true just before the loop guard is about to be evaluated (or just after the loop guard was executed as evaluating an expression here does not change any variable).

We will employ an *inductive*[1] reasoning. In order to show inductively that some statement is always true about the iterative command we will prove

1. *First time.* The statement is true when the loop is about to be executed for the first time, i.e. the loop guard is about to be evaluated for the first time.

   This step is also called *initialization* or *the base case.*

2. *Once through the loop.* Assume that the loop has been executed an arbitrary number of times and the statement we want to prove holds. After assuming that the loop guard is true (so the loop does not exit), we demonstrate that the statement is true when the loop is executed once more and the guard is about to be executed for the next time.

   This step is also called *maintenance* or *the inductive step.*

## 1.1 Invariant *I1*

***First time.*** From the preconditions we know that $0 < W$ and $0 < B$ and because they are natural numbers $1 \leq W$ and $1 \leq B$ which yield $1 < W + B$. When the loop is executed for the first time we have $w = W$, and $b = B$ and thus $1 \leq w + b$.

***Once through the loop.*** Let $w_c$ and $b_c$ be the *current* values of $w$ and $b$. Assume the loop continues its execution, i.e.

$$1 < w_c + b_c \tag{1}$$

so the loop body executes before the guard is evaluated again. After this one loop body execution, let the *next* values of $w$ and $b$ be $w_n$ and $b_n$. Our goal is to prove that $1 \leq w_n + b_n$. How do the next values relate to $w_c$ and $b_c$? This can be answered by considering three cases of the **if** command.

- Case $bn1.col = white$ & $bn2.col = white$. We have $w_n = w_c - 2$ and $b_n = b_c + 1$. Then $w_n + b_n = w_c + b_c - 1$.

- Case $bn1.col = black$ & $bn2.col = black$. We have $w_n = w_c$ and $b_n = b_c - 1$. Then $w_n + b_n = w_c + b_c - 1$.

- Case (**else**) $bn1.col \neq bn2.col$. We have $w_n = w_c$ and $b_n = b_c - 1$. Then $w_n + b_n = w_c + b_c - 1$.

From Equation (1) we have $1 - 1 < w_c + b_c - 1$ and therefore $0 < w_n + b_n$ in each case above. As $w_n + b_n$ is an integer we conclude that $1 \leq w_n + b_n$ no matter which action of the iterative command is executed.

The two steps named *First time* and *Once through the loop* demonstrate that $1 \leq w + b$ is a loop invariant as it is true no matter how many times the loop was executed.

## 1.2 Invariant *I2*

We use the same technique and notation as above.

[1]Induction will be discussed in detail later in the course.

***First time.*** Because $w = W$, then $w = W \pmod 2$.

***Once through the loop.*** (Look above for notation.)

Assume the invariant holds, i.e.

$$w_c = W \pmod 2 \tag{2}$$

and the loop does not exit. Our goal is to prove that $w_n = W \pmod 2$. We consider three cases of the **if** command.

- Case $bn1.col = white$ & $bn2.col = white$. We have $w_n = w_c - 2$ and hence $w_n = W \pmod 2$ by Equation (2) and the definition of $\pmod 2$.
- Case $bn1.col = black$ & $bn2.col = black$. We have $w_n = w_c$ and therefore $w_n = W \pmod 2$ by Equation (2).
- Case (**else**) $bn1.col \neq bn2.col$. We have $w_n = w_c$ and therefore $w_n = W \pmod 2$ by Equation (2).

The invariant labeled *I2* is now proven. The invariant says that the parity of white beans in the can never changes, i.e. the parity is an invariant of the loop.

## 1.3 The loop terminates

This claim is proven with the help of the *variant* expression. Recall that the variant expression must be a natural (integer, non-negative) expression which decreases every time the loop is executed. If so, the loop must terminate, as no natural expression can decrease forever. This is another inductive reasoning in the form of the *least element principle* for natural numbers.

In our case the variant expression is claimed to be $w + b$. Without a proof that the expression is a variant of the loop such a claim is nothing more than a guess. Many different expressions can be proven to be a variant for a terminating loop. (*Make sure that you understand this. After following this proof find some other variant expressions for our loop.*)

***First time.*** Here we have to show that $w + b$ is a natural expression. It is easy to see that this is the case as $w + b = W + B$.

***Once through the loop.*** Look into Section 1.1 for notation and assumptions.

We have to prove that $0 \leq w_n + b_n < w_c + b_c$.

From the *Once through the loop* reasoning in Section 1.1 we know that $w_n + b_n = w_c + b_c - 1$ so $w_n + b_n < w_c + b_c$. In that reasoning we also proved that $0 < w_n + b_n$, therefore $0 \leq w_c + b_c - 1$. We have proved that $w + b$ is the sought variant expression and we can conclude that the loop terminates.

Note that a variant expression for a terminating loop is not unique. In the above case take for example $w + b + 31$. It is a variant expression as well. Do you know how to prove this claim?

## 1.4 Proving the postconditions

One may ask why we picked these invariants and not other ones. The answer is simple: the invariants help in proving the claimed postconditions. But indeed, there are many formulae which are invariants for the loop and some of them are completely useless as for example the following property which is always true: $1 = 1$.

We know that the loop terminates, cf. Section 1.3. The question is now: do the postconditions hold when the loop does terminate? The answer is *yes* because of the following reasoning.

When the loop exits (it must as it terminates) we have

$$w + b \leq 1 \tag{3}$$

as this is the negation of the loop guard. We also know that the invariants hold as they held just before the loop exited.

$$1 \leq w + b \tag{4}$$

$$w = W \pmod 2 \tag{5}$$

The postcondition labeled *R1* follows directly from Equations (3) and (4). Now we know that the loop exists with one bean in the can.

The postcondition labeled *R2* is just Equation (5), so it holds. How does it answer the question about the color of the last bean? From *R1* and the fact that both $w$ and $b$ are natural numbers we know that either $(w = 1) \wedge (b = 0)$ or $(w = 0) \wedge (b = 1)$. *R2* says that $w = W \pmod 2$ and from the definition of $\pmod 2$ we know that $1 = W \pmod 2$ or $0 = W \pmod 2$.

- If $1 = W \pmod 2$ (i.e. $W$ is odd) then $w = 1$ and the last bean is white.
- If $0 = W \pmod 2$ (i.e. $W$ is even) then $w = 0$ and the last bean is black.

In words we can say: the process of removing beans from the can terminates with one bean in the can; the remaining bean is white exactly when the initial number of white beans was odd.

The essence of the argument is that the parity of white beans is an invariant of the terminating loop.

## 1.5 Various coffee cans

Consider the following three variations on the coffee can problem. In each case, replace the **if** statement in the program on p. 4 by one of the **if** statements below. For each of the variations do the following.

1. State a variant for the resulting **while**-loop.
2. Find out the color of the remaining bean by formulating relevant invariants and post-conditions or prove that the color of the last bean cannot be established.
3. Justify your claims. You do not have to be as detailed as our handling of the original coffee can problem above, but provide evidence that you know what you are doing.

**Variation A**

```
if $bn1.col = white$ & $bn2.col = white$ then
    $PutBlack$;
    $w, b := w - 2, b + 1$
elseif $bn1.col = black$ & $bn2.col = black$ then
    $PutBlack$;
    $b := b - 1$
else
    $PutBlack$;
    $w := w - 1$;
endif
```

**Variation B**

```
if $bn1.col = white$ & $bn2.col = white$ then
    $PutWhite$;
    $w := w - 1$
elseif $bn1.col = black$ & $bn2.col = black$ then
    $PutBlack$;
    $b := b - 1$
else
    $PutBlack$;
    $w := w - 1$;
endif
```

**Variation C**

```
if $bn1.col = white$ & $bn2.col = white$ then
    $PutBlack$;
    $w, b := w - 2, b + 1$
elseif $bn1.col = black$ & $bn2.col = black$ then
    $PutBlack$;
    $b := b - 1$
elseif $bn1.col = white$ & $bn2.col = black$ then
    $PutBlack$;
    $w := w - 1$
else
    $PutWhite$;
    $b := b - 1$;
endif
```

## 2 Euclid's Algorithm

*Note* For historical reasons Euclid's algorithm is the best candidate for the first encounter with algorithms. Google 'Euclid' to see why.

**Definition** Given two integer numbers $x$ and $y$, we say that $x$ divides $y$, $x \mid y$ for short, if there exists an integer $z$ such that $z \cdot x = y$. (See also [1, p. 54].)

**Definition** Given two natural numbers $x$ and $y$, at least one of them non-zero, we define the greatest common divisor of $x$ and $y$, $\gcd(x, y)$ for short, as the largest natural number that divides both $x$ and $y$. (See also [1, p. 68].)

The following 'recipe' for computing the greatest common divisor of two numbers is attributed to Euclid, ca. 325 BC–265 BC, a Greek mathematician, who lived in Alexandria, Hellenistic Egypt. We start with a simpler version of Euclid's algorithm which uses only subtraction (and no division).

*Euclid's recipe*

```
$X, Y \in \mathbb{N}$;
    read ( X, Y );
$x, y \in \mathbb{N}$;
    $x, y := X, Y$;
    while $x \neq y$ do
        if $x > y$ then
            $x := x - y$
        else
            $y := y - x$
    end
    print ( x )
```

*End of Euclid's recipe*

What you see above *is not* a program if we do not positively answer the following two questions:

- Does Euclid's recipe stop? Why?
- Is the printed value equal to $\gcd(X, Y)$? Why?

In trying to answer these questions we will have to appeal to some properties of gcd. Here is a list of some of the properties, we assume that at least one of the arguments to gcd is non-zero.

(a) $\gcd(x, y) = \gcd(y, x)$.
(b) $\gcd(x, y) = \gcd(x + y, y)$.
(c) $\gcd(x, y) = \gcd(x - y, y)$.
(d) $\gcd(x \cdot y, y) = y$.
(e) $\gcd(x, x) = x$.
(f) $\gcd(x, 0) = x$.

We should have started with the specification of the Euclid's algorithm before writing any code. Better late than never, here it is.

*Specification of Euclid's algorithm*

```
$X, Y, r \in \mathbb{N}$
          PRECONDITIONS: $X > 0$ & $Y > 0$.
          POSTCONDITIONS: $r = \gcd(X, Y)$.
```

*End of the specification of Euclid's algorithm*

Note that in the specification we do not mention the *read* and *print* commands as the essence of this algorithm is independent of the way the input parameters are taken and the output parameters are reported. For this reason we will not mention the input/output commands in our solution.

We will now present Euclid's algorithm and then argue that the algorithm implements the above specification. We include a variant expression and a relevant invariant formula in the code together with the specification.

*Euclid's algorithm*

```
$X, Y \in \mathbb{N}$;
          PRECONDITIONS $X > 0$ & $Y > 0$;
$x, y \in \mathbb{N}$; $x, y := X, Y$;
while $x \neq y$ do
                    VARIANT $x + y$.
                    INVARIANT $\gcd(x, y) = \gcd(X, Y)$.
          if $x > y$ then
                    $x := x - y$
          else
                    $y := y - x$
endwhile
POSTCONDITIONS $x = \gcd(X, Y)$.
```

*End of Euclid's algorithm*

We give our reasons why the program presented above is correct wrt (read: with respect to) the specification.

1. That the loop terminates is guaranteed by the expression named VARIANT. This expression takes on positive integer values and its value decreases every time the loop is executed. Therefore, the value must stop decreasing at certain point, which means that the loop terminates.

2. That the postconditions are satisfied is demonstrated with the help of the statement named INVARIANT about which we claim that

   (a) Before the loop is executed for the first time, the invariant holds.

   (b) If the invariant is true and the loop is executed once more, the invariant will still be true.

Therefore, the invariant is always true, by the principle of inductive reasoning.

3. The invariant is true when the loop terminates (and that it does terminate we know from having a variant). When the loop terminates we have that $x = y$ and we have the invariant $\gcd(x, y) = \gcd(X, Y)$. The properties of gcd tell us that under such conditions $\gcd(x, y) = x$ and the postconditions are clearly true.

The way we gave our justification is far from formal. However, without further ado we claim that it is *easy to see*[2] that the program has the claimed properties.

For practice, let us try to be more detailed about our claims. To set the stage for the rest of the solution we first show that $x, y > 0$ is also a loop invariant, i.e. $x, y > 0$ holds when the loop guard is about to be evaluated, i.e. just before checking whether $x \neq y$.

**Initialization** or *base case* or *first time.* Just before the first time evaluation of the guard, $x = X$ and $y = Y$ so the condition $x, y > 0$ holds as $X, Y \in \mathbb{N}^+$.

**Maintenance** or *the inductive step* or *once through the loop.* Assume that at some point of the algorithm run, $x, y > 0$ and the loop guard is true, $x \neq y$. Then the loop is executed at least once more. Let the new values of $x$ and $y$ after this iteration be $x'$ and $y'$, respectively. We consider two cases.

1) If $(x > y)$, then $x' = x - y > y - y = 0$;

2) (**else**) If $(x < y)$, then $y' = y - x > x - x = 0$.

Therefore, $x, y > 0$ is always true, i.e. it is a loop invariant.

## 2.1 Euclid's variant

$x + y > 0$ initially. If at some point of loop execution $x \neq y$, then the loop is executed (at least) once more and let the new values of $x$ and $y$ after one iteration be $x'$ and $y'$, respectively. We consider two cases:

1. If $(x > y)$, then $x' = x - y$ and $y' = y$, and thus $x' + y' = x - y + y = x$. Therefore, $x' + y' > 0$ and $x' + y' = x < x + y$, since $x, y > 0$.

2. If $(x < y)$, then $y' = y - x$ and $x' = x$, thus $x' + y' = x + y - x = y$. Therefore, $x' + y' > 0$ and $x' + y' = y < x + y$, since $x, y > 0$.

We have shown that $x + y$ is a loop variant as it is positive and its value decreases with every iteration of the loop. The loop terminates.

[2]Whenever they write that *it is easy to see that ...* it usually means that they do not know how to explain it any better (like in the case above), or they simply do not know how to explain it at all, or that the claimed thing is simply false. The poor reader is to figure out which is the case.

### 2.2 Euclid's invariant

Initially, $x = X, y = Y$, and so $\gcd(x, y) = \gcd(X, Y)$.

Now consider that at some point of the iteration $\gcd(x, y) = \gcd(X, Y)$ and the loop guard is about to be evaluated. If $x = y$ then the loop terminates. Otherwise, the loop executes and let the new values of $x$ and $y$ after one iteration be $x'$ and $y'$, respectively. We consider two cases:

1. If $(x > y)$, then $x' = x - y$ and $y' = y$. $\gcd(x', y') = \gcd(x - y, y) = \gcd(x, y) = \gcd(X, Y)$.
2. If $(x < y)$, then $y' = y - x$ and $x' = x$. $\gcd(x', y') = \gcd(x, y - x) = \gcd(x, y) = \gcd(X, Y)$.

See the discussion of the Euclidean algorithm, [1][p. 70].

The above proves that $\gcd(x, y) = \gcd(X, Y)$ is an invariant of the loop.

### 2.3 Euclid's postconditions

According to Section 2.1 the loop terminates with $x = y$. Since $x > 0$ then $\gcd(x, x) = x$. Therefore $x = \gcd(x, y) = \gcd(X, Y)$.

### 2.4 A variation on Euclid's algorithm

**Definition** Given two non-zero natural numbers $x$ and $y$, the *least common multiple* of $x$ and $y$, $\mathrm{lcm}(x, y)$ for short, is the smallest non-zero natural number that is divisible by both $x$ and $y$. (See also [1, p. 68].)

Consider the following program.

*Euclid's algorithm variation 1*

```
$a, b \in \mathbb{N}^+$;
$x, y, u, v \in \mathbb{N}$;
$x, y, u, v := a, b, a, b$;
while $(x \neq y)$ do
        VARIANT $x + y$.
        INVARIANT $\gcd(x, y) = \gcd(a, b)$ & $x \cdot u + y \cdot v = 2ab$.
    if $(x > y)$ then
        $x := x - y,\ u := u + v$;
    else
        $y := y - x,\ v := v + u$;
endwhile;
$g, \ell \in \mathbb{N}$;
$g := x,\ \ell := (u + v)/2$;
POSTCONDITIONS $g = \gcd(a, b)$ & $\ell = \mathrm{lcm}(a, b)$.
```

*End of Euclid's algorithm variation 1*

That the program terminates is clear from the original Euclid's algorithm. Using the stated invariant and properties of gcd and lcm (not stated here), prove that the above program is correct wrt the specification.

## 2.5 Extended Euclid's algorithm

So far we have been computing the gcd using only subtraction. Here we allow ourselves to use division (integer division, as did Euclid) and we also solve a more interesting problem. The program computes coefficients of linear combinations of two numbers resulting, respectively, in 0 and their gcd value. Prove that the program below is correct.

*Extended Euclid's algorithm*

```
$X, Y, r \in \mathbb{N}$;
$a, b, a1, b1 \in \mathbb{Z}$;
    PRECONDITIONS $X \geq Y > 0$.
$x, y \in \mathbb{N}$;
$a, b, a1, b1, x, y := 0, 1, 1, 0, X, Y$;
while $(y \neq 0)$ do
        VARIANT $x + y$.
        INVARIANT $\gcd(x, y) = \gcd(X, Y)$ &
                  $0 \leq y \leq x$ &
                  $a \cdot X + b \cdot Y = y$ & $a1 \cdot X + b1 \cdot Y = x$.
    $a, b, a1, b1 := a1 - (x \,/\, y) \cdot a, b1 - (x \,/\, y) \cdot b, a, b$
    $x, y := y, x \,\%\, y$;
endwhile
$r := x$
POSTCONDITIONS $r = \gcd(X, Y)$ & $a \cdot X + b \cdot Y = 0$ & $a1 \cdot X + b1 \cdot Y = r$.
```

*End of extended Euclid's algorithm*

## 2.6 A problem

Replace the question marks below to make the program correct and provide a proof.

*Extended slow Euclid's algorithm*

```
$X, Y, r, a, b \in \mathbb{N}$;
    PRECONDITIONS $X > 0$ & $Y > 0$.
$x, y, a1, b1 \in \mathbb{N}$;
$a, b, a1, b1, x, y := 1, 0, 0, 1, X, Y$;
while $(x \neq y)$ do
        VARIANT $x + y$.
        INVARIANT $\gcd(x, y) = \gcd(X, Y)$ &
                  $a \cdot X + b \cdot Y = x$ & $a1 \cdot X + b1 \cdot Y = y$.
    if $x > y$ then
        $x, a, b := x - y, ?, ?$
    else
        $y, a1, b1 := y - x, ?, ?$
endwhile
$r := x$
POSTCONDITIONS $r = \gcd(X, Y)$ & $a \cdot X + b \cdot Y = r$.
```

*End of extended slow Euclid's algorithm*

# 3 Logarithm

Consider the following incomplete code where $A$ is an input parameter and $d$ is an output parameter.

*Logarithm base 3*

```
$A,\ d \in \mathbb{N}$;
        PRECONDITIONS $0 < A$.
$x \in \mathbb{N}$;
$d, x := 0, 1$;
while $(3 \cdot x \le A)$ do
                VARIANT ?
                INVARIANT $x \le A\ \ \&\ \ x = 3^d$.
        $d,\ x := d+1,\ 3 \cdot x$
endwhile
POSTCONDITIONS $d \le \log_3 A < d+1$.
```

*End of Logarithm base 3*

(*Note.* This problem is an example of a typical exam question.)

Below, we employ the standard practice of using $d'$ and $x'$ for the values of variables $d$ and $x$ after the loop was executed once and just before we are testing the loop condition again. In our case we have $d' = d + 1$ and $x' = 3 \cdot x$. Because $A$ never changes we have $A = A'$.

## 3.1 Variant

Find a variant expression for the loop and prove the properties required for calling it a variant.

$A - x$ is a variant for the loop.

*Variant is non-negative* Initially, $x = 1$, so $A - x = A - 1 \ge 0$ because of preconditions. Assume that $A - x \ge 0$ and consider one loop execution when $3 \cdot x \le A$. We have that $x' = 3 \cdot x$ and because of the assumed loop guard we have $A' - x' \ge 0$.

*Variant decreases with each loop execution* Assume that $A - x \ge 0$ and consider one loop execution when $3 \cdot x \le A$. Since $x' = 3 \cdot x$, we would have $A' - x' < A - x$ but only if we know that $x > 0$.

Proving that $x > 0$ is a loop invariant for the loop is left as an exercise to. Alternatively, one can appeal to the second conjunct of the stated (but yet not proven) invariant.

## 3.2 Initialization

Show that the invariant is true upon entering the loop.

Initially, $d = 0$ and $x = 1$. By preconditions $A > 0$, so by arithmetic $A \ge 1$ and by algebra $1 = 3^0$.

## 3.3 Maintenance

Show that if the invariant is true and the loop does not exit then the invariant will stay true.

Assume the loop invariant $x \leq A \ \& \ x = 3^d$ and that the loop does not exit, so $3 \cdot x \leq A$. $x' = 3 \cdot x \leq A = A'$ follows directly from the loop guard being true. $x' = 3 \cdot x = 3 \cdot 3^d = 3^{d+1} = 3^{d'}$.

## 3.4 Postconditions

Prove that the postconditions are satisfied when the loop exits.

When the loop exits then $3 \cdot x > A$ but the invariant holds. Therefore $x \leq A < 3 \cdot x$ and so $3^d \leq A < 3 \cdot 3^d$. Since logarithm is a monotone function we have that $d \leq \log_3 A < d+1$.

## 4 Interpolation Search

We assume that you have heard of the *binary* search. Here is another, (marginally) better at times algorithm which searches a sorted array of integers $a$, indexed from 1 to $s$, for a value $x$. (The pre- and post- conditions specify in precise terms what the program is expected to do.)

*Interpolation search*

```
$s \in \mathbb{N}^+,\ A \in \mathbb{Z}^s,\ x \in \mathbb{Z},\ found \in \{F, T\};$
   PRECONDITIONS $\forall i \in [1..s), A[i] < A[i+1]$.
$h, \ell, r \in \mathbb{N};$
$\ell, r, found := 1, s, F;$
while $(\sim found \wedge (\ell \leq r))$ do
   if $((x < A[\ell]) \vee (A[r] < x))$ then
      $\ell := r + 1$
   else if $(\ell = r)$ then
      $h, found := \ell, T$
   else
      $h := \ell + \left\lfloor (r - \ell) \times \frac{x - A[\ell]}{A[r] - A[\ell]} \right\rfloor;$
      if $(A[h] = x)$ then
         $found := T$
      else if $(A[h] > x)$ then
         $r := h - 1$
      else
         $\ell := h + 1$
      endif
   endif
endwhile
POSTCONDITIONS $(found \Rightarrow (A[h] = x)) \wedge (\sim found \Rightarrow (\forall i \in \{1..s\}, A[i] \neq x))$.
```

*End of interpolation search*

In our discussion of the program we will use a variable $f \in \mathbb{N}$ defined to be $found ? 1 : 0$ (i.e. $f$ is 1 when *found* is true, otherwise $f$ is 0).

### 4.1 Variant

We show that $V = 1 + r - \ell - f$ is a loop variant, i.e. $V$ is always non-negative and decreases with every loop iteration. Initially, $\ell = 1$, $r = s$, $f = 0$, so $V = 1 + s - 1 - 0 = s + 1 > 0$, since $s \in \mathbb{N}^+$.

Assume loop condition is true (i.e., $f = 0 \wedge l \leq r$), therefore $0 < V$. Let $\ell', r', f', V'$ be the (new) values of $\ell, r, f, V$ after one execution of the loop body, respectively. We want to prove that $V > V' \geq 0$.

- **if** $((x < A[\ell]) \vee (A[r] < x))$

  $\ell' = r + 1$, $r' = r$, $f' = f = 0$. So $V' = 1 + r - r - 1 - f = 0$. Since $V > 0$, we have $V > V' \geq 0$.

- **else if** $(\ell = r)$:
  $\ell' = \ell, r' = r, f' = 1$. So $V' = 1 + r' - \ell' - f' = V - 1$. Since $V > 0$, $V' > V - 1 \geq 0$.
- **else** (We must have here $l < r \wedge A[\ell] \leq x \leq A[r]$.)
  $0 \leq \frac{x - A[\ell]}{(A[r] - A[\ell])} \leq 1$ and hence, $0 \leq \left\lfloor (r - \ell) \times \frac{x - A[\ell]}{A[r] - A[\ell]} \right\rfloor \leq r - \ell$. Thus, $\ell \leq h \leq r$.
  - **if** $(A[h] = x)$
    $\ell' = \ell, r' = r, f' = 1$. So $V' = 1 + r' - \ell' - f' = V - 1$. Since $V > 0$, $V' > V - 1 \geq 0$.
  - **else if** $(A[h] > x)$
    $\ell' = \ell$, $r' = h - 1$, $f' = f$. Since $h \leq r$, we have $r' = h - 1 \leq r - 1 < r$. Thus, $V > V'$. Since $A[h] > x$ and $x \geq A[l]$, we have $A[h] > A[l]$, which means $h > l$. Therefore, $r' = h - 1 \geq \ell = \ell'$. So, $V' = 1 + r' - \ell' - f' \geq 1 - f' \geq 0$.
  - **else** (We must have here $A[h] < x$.)
    $\ell' = h + 1$, $r' = r$ and $f' = f$. Since $\ell \leq h$, we have $\ell < h + 1 = \ell'$. Thus, $V > V'$. Since $x \leq A[r]$ and $A[h] < x$, we have $A[h] < A[r]$, which means $h < r$. Therefore, $\ell' = h + 1 \leq r = r'$. So, $V' = 1 + r' - \ell' - f' \geq 1 - f' \geq 0$.

We have shown above that $V \geq 0$ is a loop invariant and also that $V$ is a loop variant.

## 4.2 A helpful invariant

We prove that the following formula is the loop invariant

$$(\mathit{found} \Rightarrow (A[h] = x)) \wedge (\sim\!\mathit{found} \Rightarrow (\forall i \in \{1..\ell - 1\} \cup \{r + 1..s\}, A[i] \neq x)).$$

Our proof is by induction on the number of iterations of the loop.

Throughout the proof we will use $Q(c, d)$ to mean $\{1..c-1\} \cup \{d+1..s\}$. $Q(c, d)$ is a set of indices in $A$ denoting the set of indices in $A$ where the sought value $x$ does **not** appear.

Let $p(n)$ be another name for the loop invariant after the **while**-loop has already iterated $n$ times and the loop guard is about to be executed.

**Initialization or the base case** After the initialization stage, when the execution just enters the **while**-loop we have that $\ell = 1, r = s, \mathit{found} = F$. The loop has been executed 0 times so far and the loop guard is about to be evaluated for the first time. We will show that $p(0)$ holds. Indeed,

- $\mathit{found} \Rightarrow (A[h] = x)$ holds as the antecedent is false.
- $\sim\!\mathit{found} \Rightarrow (\forall i \in \{1..\ell - 1\} \cup \{r + 1..s\}, A[i] \neq x)$ holds as the antecedent is true and the consequent is vacuously true because $Q(\ell, r) = \emptyset$.

**Maintenance or the inductive step** Assume we are evaluating the loop guard for the $n$-th time and (IH:) $p(n)$ holds. The loop guard evaluates to true. We will show that $p(n+1)$ holds, i.e. the loop invariant still holds when the loop guard is evaluated for the next time.

If the loop guard is true, then (A1:) $\mathit{found} = F$ and (A2:) $\ell \leq r$.

Because of (A1) and (IH) we have (B:) $\forall i \in \{1..\ell - 1\} \cup \{r + 1..s\}, A[i] \neq x$.

Let $\ell, r, h$ and *found* be the current values of the variables and let $\ell', r', h'$ and $\mathit{found}'$ be the values of the corresponding variables after executing the loop body one more time, i.e. when we deal with $p(n + 1)$ which is

$$(\mathit{found}' \Rightarrow (A[h'] = x)) \wedge ({\sim}\mathit{found}' \Rightarrow (\forall i \in \{1..\ell' - 1\} \cup \{r' + 1..s\}, A[i] \neq x)).$$

- **if** $((x < A[\ell]) \vee (A[r] < x))$

  In this case, we conclude that $x$ is not in $A[\ell..r]$. We extend the set $Q(\ell, r)$ to include also indices in the range $\{\ell..r\}$. With $\ell' = r + 1, r' = r, \mathit{found}' = \mathit{found}$ and (B) we have $\forall i \in \{1..\ell'\} \cup \{r + 1..s\}, A[i] \neq x$ and $p(n + 1)$ holds.

- **else if** $(\ell = r)$

  If this happens, then ${\sim}((x < A[\ell]) \vee (A[r] < x))$, i.e. $(x \geq A[\ell]) \wedge (A[r] \geq x)$ and with $\ell' = r$, we have $A[\ell'] = x$. With $\mathit{found}'$ set to true and $h' = \ell'$ the loop invariant $p(n + 1)$ holds.

- **else**

  If we are here then $(x \geq A[\ell]) \wedge (A[r] \geq x)$ and $\ell \neq r$. The index $h'$ of our next guess is computed.

  - **if** $(A[h'] = x)$
    $\mathit{found}'$ is true and $A[h'] = x$, so the loop invariant $p(n + 1)$ holds.
  - **else if** $(A[h'] > x)$
    In this case $x$ is not in $A[h'..r]$ as $A$ is sorted, so $Q(\ell, r)$ is extended. With $\ell' = \ell, r' = h' - 1, \mathit{found}' = \mathit{found}$ and because of (A1) and (B) the loop invariant $p(n + 1)$ holds.
  - **else**
    Here neither $(A[h'] = x)$ nor $(A[h'] > x)$. Therefore, $(A[h'] < x)$ and $Q(\ell, r)$ can be extended. With $\ell' = h' + 1, r' = r, \mathit{found}' = \mathit{found}$ and because of (A1) and (B) the loop invariant $p(n + 1)$ holds.

## 4.3 Postconditions

Our selection of loop invariant has been geared towards proving the post-conditions. We know that the loop terminates from part (a) and from part (b) we know that the invariant held when the loop guard was evaluated for the last time. There are two cases that cause the loop to exit (loop guard evaluated to false).

- if *found* is true, then the loop invariant implies the postcondition.
- Otherwise $\ell > r$ implies that $Q(\ell, r) = \{1..\ell - 1\} \cup \{r + 1..s\}$ becomes $\{1..s\}$. Thus by the loop invariant, we have ${\sim}\mathit{found} \Rightarrow (\forall i \in \{1..s\}, A[i] \neq x)$.

## 5 Zeroing a Rectangle

> There is a positive integer in each square of a rectangular table. In each move, you may double each number in a row or subtract 1 from each number in a column. Prove that you can reach a table of zeroes by a sequence of such moves. See [4, p. 9, problem 8].

Consider the following procedure to convert a column to all zeros.

```
while (not all entries in the column are 0) do
    if (some entries in the column are 1 and some are not 1)
        then double all the rows with entry 1 in the column.
    endif
    Subtract 1 from all entries in the column.
end
```

Note that after each iteration of the loop, the maximum value in the column decreases by 1. The maximum value is a non-negative integer at the start of the procedure, it is a non-negative integer throughout the procedure and therefore its value can decrease only a fixed number of times.

Note also that when the loop is executed all entries in the column are positive until all of them become 1 and then all become 0.

Suppose we run the procedure on the leftmost column and make all its entries zero. We can repeat the procedure on (any of) the remaining columns because the only time we are modifying values outside of the column is when we are doubling entries in an entire row. Since all the values in the already processed columns are zero, doubling entries in a row will not affect any solved column. Thus, we can apply the procedure to each column separately and obtain a table with all zero cells.

It is probably easier to think about solving the problem by starting at the leftmost column and performing a sweep to the right.

## 6 Potato Chips

> There are $a$ white, $b$ black, and $c$ red chips on a table. In one step, you may choose two chips of different colors and replace them by a chip of the third color. We are interested in reaching a state when there is only one chip on the the table. When is this possible? What is the color of the last chip? See [4, p. 9, problem 6].

Let us first characterize the relationship between the numbers of chips before and after a move. Let $a, b, c$ be the numbers of chips before and $a', b', c'$ be the numbers of chips after a move. A move is possible when at least two numbers among $a, b, c$ are non-zero. When we remove two chips of distinct colors and add a chip of the third color, we accomplish two things: we decrease the total number of chips by 1 and we change the *parity* of the number of chips of each color. We write E for *even* and O for *odd*.

$$a' + b' + c' = a + b + c - 1 \tag{6}$$

$$(\mathrm{O}(a) \Leftrightarrow \mathrm{E}(a')) \wedge (\mathrm{O}(b) \Leftrightarrow \mathrm{E}(b')) \wedge (\mathrm{O}(c) \Leftrightarrow \mathrm{E}(c')) \tag{7}$$

Statement (6) tells us that with each move, we decrease the total number of chips by 1. Statement (7) is the invariant property of each move and thus the entire process.

We start with $a, b, c > 0$ and want to reach one of the desired states

$$(a = 1, b = 0, c = 0), \ \ (a = 0, b = 1, c = 0) \ \text{ or } \ (a = 0, b = 0, c = 1)$$

corresponding to a white, black or red chip left on the table, respectively. When is this possible? The crucial observation is that in each desired state the parity of one of the numbers is different than the parities of the other two.

Let us focus on reaching the state $(a = 1, b = 0, c = 0)$; the other cases are analogous. In the desired state $(a = 1, b = 0, c = 0)$ the parity of $a$ is different than the parities of $b$ and $c$. If in the initial state the parity of $a$ is different than the parities of $b$ and $c$, then because of invariant (7), this property will hold after each move until we reach our desired state. However, we can get stuck if in the process we are left with chips of one color on the table before reaching the final state. With the initial condition $a, b, c > 0$ we can avoid this situation by having chips of at least two colors on the table until the last move when $(a = 0, b = 1, c = 1)$ leads to $(a = 1, b = 0, c = 0)$.

Note that if the parities of each of the initial $a, b$ and $c$ are the same, then we cannot reach any of the desired states as no matter what we do there will be at least two chips on the table (we cannot remove all the chips).

We can summarize the above reasoning as follows, if initially $a, b, c > 0$, then

$$[\mathrm{O}(a) \wedge \mathrm{O}(b) \wedge \mathrm{O}(c)] \vee [\mathrm{E}(a) \wedge \mathrm{E}(b) \wedge \mathrm{E}(c)] \Rightarrow \mathrm{unsolvable}(a, b, c) \tag{8}$$

$$[\mathrm{O}(a) \wedge \mathrm{E}(b) \wedge \mathrm{E}(c)] \vee [\mathrm{E}(a) \wedge \mathrm{O}(b) \wedge \mathrm{O}(c)] \Rightarrow \mathrm{final\_color}(\mathit{white}) \tag{9}$$

$$[\mathrm{E}(a) \wedge \mathrm{O}(b) \wedge \mathrm{E}(c)] \vee [\mathrm{O}(a) \wedge \mathrm{E}(b) \wedge \mathrm{O}(c)] \Rightarrow \mathrm{final\_color}(\mathit{black}) \tag{10}$$

$$[\mathrm{E}(a) \wedge \mathrm{E}(b) \wedge \mathrm{O}(c)] \vee [\mathrm{O}(a) \wedge \mathrm{O}(b) \wedge \mathrm{E}(c)] \Rightarrow \mathrm{final\_color}(\mathit{red}) \tag{11}$$

Together, expressions (8), (9), (10), and (11) describe every possible case for the parities of initial $a, b$ and $c$, and state when the problem is solvable, and if so, what the color of the final chip will be.

## 7 Dragon

A dragon has 100 heads. A knight can cut off 15, 17, 20, or 5 heads with one blow of his sword. In each of these cases, 24, 2, 14, or 17 new heads, respectively, grow on its shoulders. If all heads are cut off then the dragon dies. Can the dragon ever die? See [4, p. 13, problem 51].

Just before the blow that could kill the dragon, the number of dragon's heads must be 15, 17, 20, or 5. Any blow by the knight changes the number of dragon's heads by a multiple of 3, by 9, -15, -6 and 12, respectively in each case. Therefore, the initial 100 heads will never be reduced by 85, 83, 80, or 95 which are not multiples of 3. Thus the dragon never dies.

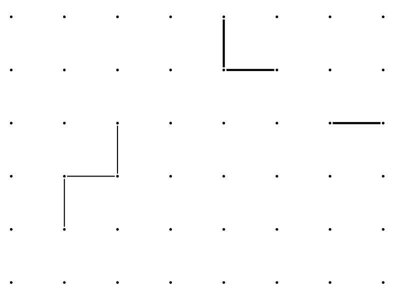
*Put shortly*: the number of dragon's heads is an invariant modulo 3. With $100 = 1 \pmod 3$ heads initially, the dragon never dies. To see this try to imagine the last, killing blow. We would need to have 15, 17, 20, or 5 heads to kill the dragon with one blow but none of these numbers is $1 \pmod 3$.

# 8 Problems to Try

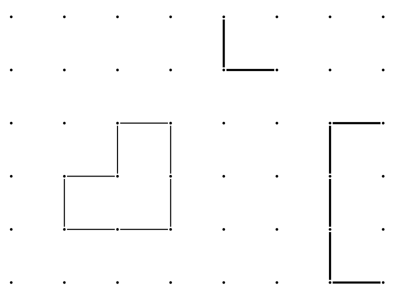
## 8.1 Closing the curve

Consider a grid of dots of any size.

Two players, $A$ and $B$, play the following game, see [5, p. 166, p. 304]. The players alternate moves, with $A$ moving first. $A$ moves by drawing a *thin* horizontal or vertical line between two adjacent dots; $B$ moves by drawing a *thick* horizontal or vertical line between two adjacent dots. For example, after three full moves the grid might be as below. A player must not write over the other player's line.



$A$ wins the game if $A$ can get a completely closed curve as shown below. $B$ has an easier task: $B$ wins if $B$ can stop $A$ from getting a closed curve.



Is there a strategy that guarantees a win for either $A$ or $B$, no matter how big the board is?

[*Hint*: What does it take to close a curve?]

## 8.2 Corn chips

There are $a$ white, $b$ black, and $c$ red chips on a table. In one step, you may choose two chips of different colors and replace each of them by a chip of the third color. Under which conditions does the process terminate, i.e. all remaining chips are of the same color? See [4, p. 9, problem 7].

Consider a case of 13 white, 15 black, and 17 red chips. What states can be reached from this starting state?

[*Hint*: Differences and sums modulo something.]

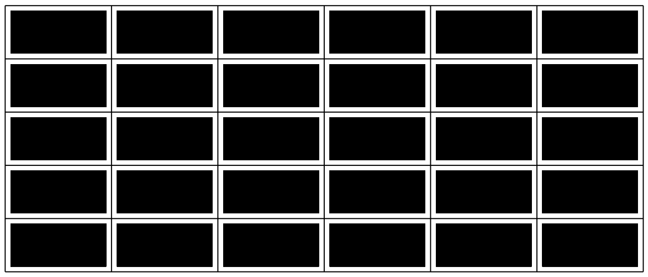
## 8.3 Matching points

There are $n$ blue and $n$ red points in the plane, no three of them collinear. Your task is to match blue points to red points by drawing matching straight line segments, each connecting one blue point to one red point such that all points are used in the matching. Show that the matching segments can be drawn with no intersections. See [3, problem 2].

[*Hint*: What can we do when two matchings cross? ]

## 8.4 Dark chocolate bar

This ridiculously easy puzzle has been known to stump some *very* high-powered mathematicians for as much as a full day, until the light finally dawns amid groans and beatings of heads against walls. See [7, p. 82].

> You have a rectangular chocolate bar marked into $m \times n$ squares, and you wish to break up the bar into its constituent squares. At each step, you may pick up one piece and break it along any of its marked vertical or horizontal lines. However, you must not stack two pieces and break them together.
>
> 
>
> Prove that every method finishes in the same number of steps.

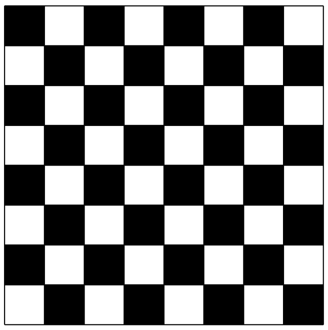
[*Hint*: In [7, p. 93] we read.

> This ridiculously easy puzzle has been known to stump some *very* high-powered mathematicians for as much as a full day, until the light finally dawns amid groans and beatings of heads against walls.]

Does your approach work for bars of white chocolate, eh?

## 8.5 Over-painting a chessboard

Assume an $8 \times 8$ chessboard with the usual coloring. You may flip the colors of all squares in (a) a row or a column, (b) a $2 \times 2$ square.



The goal is a chessboard with just one black square. Can you reach the goal? See, [4, p. 9, problem 3].

[*Hint*: (a) 8, (b) 4.]

## 8.6 Non-negative rows and columns

Start with an $m \times n$ table of integers. In one step, you may reverse the signs of all numbers in any row or column. Show that you can achieve a nonnegative sum of any row or column. See [4, p. 11, problem 24].

$$
\begin{array}{rrrrrr}
 & \downarrow & & & & \\
\hline
\multicolumn{1}{|r|}{2} & \multicolumn{1}{r|}{-3} & \multicolumn{1}{r|}{-1} & \multicolumn{1}{r|}{0} & \multicolumn{1}{r|}{3} & 1 \\
\hline
\multicolumn{1}{|r|}{-1} & \multicolumn{1}{r|}{2} & \multicolumn{1}{r|}{-1} & \multicolumn{1}{r|}{-1} & \multicolumn{1}{r|}{2} & 1 \\
\hline
\multicolumn{1}{|r|}{1} & \multicolumn{1}{r|}{3} & \multicolumn{1}{r|}{2} & \multicolumn{1}{r|}{-4} & \multicolumn{1}{r|}{2} & 4 \\
\hline
\multicolumn{1}{|r|}{2} & \multicolumn{1}{r|}{-1} & \multicolumn{1}{r|}{0} & \multicolumn{1}{r|}{3} & \multicolumn{1}{r|}{1} & 5 \\
\hline
\multicolumn{1}{|r|}{-3} & \multicolumn{1}{r|}{-2} & \multicolumn{1}{r|}{-2} & \multicolumn{1}{r|}{4} & \multicolumn{1}{r|}{4} & 1 \\
\hline
1 & -1 & -2 & 2 & 12 &
\end{array}
\quad \Longrightarrow \quad
\begin{array}{rrrrrr}
\hline
\multicolumn{1}{|r|}{2} & \multicolumn{1}{r|}{3} & \multicolumn{1}{r|}{-1} & \multicolumn{1}{r|}{0} & \multicolumn{1}{r|}{3} & 7 \\
\hline
\multicolumn{1}{|r|}{-1} & \multicolumn{1}{r|}{-2} & \multicolumn{1}{r|}{-1} & \multicolumn{1}{r|}{-1} & \multicolumn{1}{r|}{2} & -3 \\
\hline
\multicolumn{1}{|r|}{1} & \multicolumn{1}{r|}{-3} & \multicolumn{1}{r|}{2} & \multicolumn{1}{r|}{-4} & \multicolumn{1}{r|}{2} & -2 \\
\hline
\multicolumn{1}{|r|}{2} & \multicolumn{1}{r|}{1} & \multicolumn{1}{r|}{0} & \multicolumn{1}{r|}{3} & \multicolumn{1}{r|}{1} & 7 \\
\hline
\multicolumn{1}{|r|}{-3} & \multicolumn{1}{r|}{2} & \multicolumn{1}{r|}{-2} & \multicolumn{1}{r|}{4} & \multicolumn{1}{r|}{4} & 5 \\
\hline
1 & 1 & -2 & 2 & 12 &
\end{array}
$$

[*Hint*: What increases? Is there a bound on the number of reachable states?]

## 8.7 Solitaire of stepping stones

$n$ boxes are located at the perimeter of a circle and numbered 1 to $n$ in clockwise order. Each of the boxes contains more than $n$ stones, In the first move, all stones from box 1 are removed and put one by one into the following boxes starting with box 2 in clockwise direction. Now, all stones (if any) in box 1 are removed and thrown away. An analogous move is now performed with box 2, then with box 3, and so on. The moves are continued until after having distributed all the stones from a box, there are no stones to throw away. See [2, p. 53].

(a) How many stones are left when the game stops?

(b) How many moves are performed?

(c) What is the final arrangement of stones in boxes.

## 8.8 Cannot go all positive

In the following table

| | | | |
|---|---|---|---|
| 1 | 1 | 1 | 1 |
| $-1$ | 1 | 1 | 1 |
| 1 | 1 | 1 | 1 |
| 1 | 1 | 1 | 1 |

you may switch the signs of all numbers of a row, column, or a parallel to one of the diagonals. In particular you can switch the sign of each corner square. Prove that always at least one $-1$ will remain in the table. See [4, p. 9, problem 11].

[*Hint*: Look at the borders that are not corners.]

## 8.9 Infection

Nine $1 \times 1$ squares of a $10 \times 10$ board are infected. In one time unit, the cells with at least two infected neighbor cells (having a common side) become infected. Can the infection spread to the entire board? See [4, p. 12, problem 43], [7, p. 79], [8, p. 91].

[*Hint*: What "measure" of infected cells does not increase?]

Ponder this

(a) What is the minimum number of infected cells on a $m \times n$ board such that the infection can spread to the entire board? Show at least two different starting states.

(b) What is the maximum number of infected cells such that the infection cannot spread to the entire board $m \times n$?

(c) Consider $d$-dimensional $n \times n \times \cdots \times n$ hypercube. If a unit hypercube has $d$ or more infected neighbors, then it becomes infected itself. Prove that you can infect the entire hypercube starting with just $n^{d-1}$ sick unit hypercubes.

### 8.10 Row under row and so on

There is a row of $n$ integers. Just below it we write another row constructed as follows. Under each number $a$ of the first row we write a positive integer equal the number of occurrences of $a$ in the first row. In an analogous way, we get a 3rd row from the 2nd row, and so on. Prove that there is a $k$ such that rows $k$ and $k+1$ are identical. See [4, p.10, problem 12].

[*Hint*: Do a few examples and you shall see.]

### 8.11 Now to 3D

Seven vertices of a cube are marked 0 and one is marked by 1. You may select an edge and increase by one the numbers at the ends of that edge. While repeating the step your goal is to reach (a) 8 equal numbers, (b) 8 even numbers, (c) 8 numbers divisible by 3. Can you do it? See [4, p. 14, problem 59].

[Not much of a hint for (c): draw a projection of the cube with no intersecting line segments and divide the vertices into two groups].

### 8.12 Who wins?

A row of minuses is written on a piece of paper. Two players take turns in replacing either a single minus or two adjacent minuses by pluses. The one who cannot make the move loses. Is there a winning strategy? See [9, p. 1].

[*Hint*: Symmetry.]

### 8.13 Parliament

In a parliament, each MP has at most three enemies and being an enemy is a mutual feeling. How to divide the parliament into two chambers such that no parliamentarian has more than one enemy in their chamber? See [9, p. 1].

[*Hint*: Divide and move.]

### 8.14 1G of numbers

Each of the numbers from 1 to $10^9$ is repeatedly replaced by its digital sum until we reach $10^9$ one digit numbers. Will there be more 1's or 2's? See [4, p. 9, problem 9].

[*Hint*: 9.]

### 8.15 Divisible by 4?

Each of the numbers $a_1, a_2, \ldots, a_n$ is 1 or $-1$ and we form

$$a_1a_2a_3a_4 + a_2a_3a_4a_5 + \cdots + a_na_1a_2a_3 = 0$$

Prove that $4 \mid n$. See [4, p. 5, problem E7].

[*Hint*: What happens to the sum if we replace $a_i$ by $-a_i$?]

## 8.16 Shrinking quadruples

Start with a sequence $S$: $S_0 = (a_0, b_0, c_0, d_0)$ and given $S_n = (a_n, b_n, c_n, d_n)$ let

$$S_{n+1} = (|a_n - b_n|, |b_n - c_n|, |c_n - d_n|, |d_n - a_n|)$$

. Is there a $k$ such that $S_k = (0, 0, 0, 0)$? See [4, p. 7, problem E10].

Some practice:

- $(0, 3, 10, 13) \mapsto \ldots$
- $(8, 17, 3, 107) \mapsto \ldots$
- $(91, 108, 95, 294) \mapsto \ldots$

Some questions:

- What can you say about the sequence $\max S$?
- How different are behaviors of $S$ and $tS$?
- After how many steps do all four terms become even? divisible by $2^2$? by $2^k$?

## 8.17 Keep the sum intact

Consider the following process.

*Keep the sum intact*

```
$a, b \in \mathbb{N}^+$;
while $a \neq b$ do
        VARIANT ?
    if $a < b$ then
        $a, b := 2a, b - a$
    else
        $a, b := a - b, 2b$
endwhile
```

*End of Keep the sum intact*

An interesting invariant (if you need one) is easy to guess from the name of the problem.

For which starting $a$ and $b$ does the algorithm stop? In how many steps does it stop, if it stops? What can you tell about period and tails? See [4, p. 9, problem 4].

[*Hint*: Binary.]

### 8.18 The Collatz Problem: harder than it looks

The following problem seems to be very hard. The solution is unknown as of this writing. Can you find a variant for the loop in the code below? If you can, tell us immediately.

*The 2, 3 problem*

```
$x \in \mathbb{N}$;
    PRECONDITIONS $x > 0$.
while $x \neq 1$ do
        VARIANT ?
        INVARIANT $x > 0$.
    if $odd(x)$ then
        $x := 3 \cdot x + 1$
    else
        $x := x \;/\; 2$
endwhile
POSTCONDITIONS $x = 1$.
```

*End of the 2, 3 problem*

## References

[1] E. A. Bender and S. G. Williamson, *A Short Course in Discrete Mathematics*, Dover, 2005.

[2] G. Cohen, editor, *50 Mathematical Puzzles and problems, Red Collection*, Key Curriculum Press, 2001.

[3] E. W. Dijkstra, *Reasoning about Programs*, Sun Microsystems, 1990 (VHS tape).

[4] A. Engel, *Problem-Solving Strategies*, Springer-Verlag, New York, 1998.

[5] D. Gries, *The Science of Programming*, Springer-Verlag, New York, 1981.

[6] A. Liu, *Personal communication*, Edmonton, 2008.

[7] P. Winkler, *Mathematical Puzzles: a connoisseur's collection*, A K Peters 2004.

[8] P. Winkler, *Mathematical Mind-Benders*, A K Peters 2007.

[9] S. Savchev and T. Andreescu, *Mathematical Miniatures*, MAA 2003.