CS 577: Introduction to Algorithms Fall 2015

Program Correctness

Instructor: Dieter van Melkebeek Scribe: Dieter van Melkebeek

In these notes we discuss the notion of program correctness. We illustrate how to argue program correctness for simple iterative and recursive procedures, using loop invariants and recursion trees, respectively. Along the way we review some mathematical notation and concepts, including proofs by induction.

1 Program Specification and Correctness

It is important to distinguish between a program and its *specification*. The program specification contains all the information a user needs to know and can rely upon regarding *what* the program does, abstracting away details of *how* the program does it.

In its most basic form, a program specification is a precise description of the computational problem the program solves, *i.e.*, *what* relation between inputs and outputs the program realizes. A basic program specification consists of the following parts:

Input: A description of the *valid inputs*: what types they are, and possibly also what additional restrictions they are supposed to satisfy.

Output: A description of the *correct output*: for each valid input, what the output is supposed to be.

The additional restrictions on the input are referred to as the *preconditions* of the program. A program should only be called with inputs that satisfy the preconditions; the behavior of the program on other inputs is unspecified. The correct output on valid inputs need not be unique. In mathematical terms, the relation between valid inputs and correct outputs need not be a mapping from valid inputs to outputs.

If a program has side effects, those should also be part of the specification. Other properties, such as bounds on the running time, may also be included in the specification.

A program is deemed *correct* if it satisfies its specification. For the above basic form of program specification, this means that for *all* inputs that satisfy the preconditions, the program returns a correct output as prescribed in the program specification.

Arguing program correctness often happens in two steps:

- Partial correctness: For every valid input, assuming the program terminates on that input, it produces a correct output for that input.
- Termination: The program terminates on every valid input.

In the remainder of these notes, we illustrate the approach for simple iterative and recursive procedures for two computational problems on integers: powering and computing greatest common divisors.

2 Powering

We specify the Powering Problem and use the opportunity to review some math background. We start with some basics about sets and numbers.

Sets A set is a collection of elements. The collection may be finite or infinite. Finite sets can be described by listing their elements; we write them between curly braces; *e.g.*, $\{1, 5, 7\}$ denotes the set consisting of the integers 1, 5, and 7. The order of the elements does not matter; *e.g.*, $\{1, 5, 7\} = \{7, 1, 5\}$. Similarly, $\{\}$ denotes the set with no elements; we call $\{\}$ the empty set and also denote it by $\varnothing$. For a given element s and set S, we say that s belongs to S and write "$s \in S$" if s is an element of S; we write"$s \notin S$" otherwise.

For two sets S and T, we write $S \subseteq T$ to denote that S is a subset of T; *i.e.*, every element of S is also an element of T. We write $S \not\subseteq T$ to denote that S is not a subset of T; *i.e.*, some element of S is not an element of T.

Operations on sets The intersection of S and T, written $S \cap T$, is the set consisting of all elements that belong to both S and T:

$$S \cap T \doteq \{x \text{ such that } x \in S \wedge x \in T\}$$

where we write a dot on top of the equality sign to indicate that the equality holds by definition, and $\wedge$ denotes "and". We say that S and T are disjoint if $S \cap T = \varnothing$.

The union of S and T, written $S \cup T$, is the set consisting of all elements that belong to at least one of S and T:

$$S \cup T \doteq \{x \text{ such that } x \in S \vee x \in T\}$$

where $\vee$ denotes "or".

The difference of S with T, denoted $S \setminus T$, is the set consisting of all elements that belong to S but not to T:

$$S \setminus T \doteq \{x \text{ such that } x \in S \wedge x \notin T\}.$$

The Cartesian product of S with T, denoted $S \times T$, is the set consisting of all ordered pairs consisting of an element of S followed by an element of T:

$$S \times T \doteq \{(x, y) \text{ such that } x \in S \wedge y \in T\}.$$

Note the use of the regular parentheses in the notation (x, y) for ordered pairs. Whereas $\{x, y\} = \{y, x\}$ always holds, $(x, y) \neq (y, x)$ unless $x = y$. A relation of S with T is a subset $R \subseteq S \times T$. A mapping from S to T is a relation of S with T such that for every $x \in S$ there is exactly one $y \in T$ satisfying $(x, y) \in R$. S is called the domain of the mapping, and T to codomain.

Numbers An example of an infinite set is set of the integers $\{\ldots, -3, -2, -1, 0, 1, 2, 3, \ldots\}$, which we denote by $\mathbb{Z}$ (the first letter of "Zahl", the German word for "number"). We write $\mathbb{N}$ for the set of natural numbers $\{0, 1, 2, 3, \ldots\}$, and $\mathbb{Z}^+$ for the set of positive integers $\{1, 2, 3, \ldots\}$. We also use the set of rational numbers $\{\frac{a}{b}$ such that $a \in \mathbb{Z} \wedge b \in \mathbb{Z}^+\}$ denoted by $\mathbb{Q}$ (the first letter of "quotients"), the set $\mathbb{R}$ of reals, and the set $\mathbb{R}^+$ of positive reals. We have that

$$\mathbb{N} \subseteq \mathbb{Z} \subseteq \mathbb{Q} \subseteq \mathbb{R}$$

Definition and specification We specify the Powering Problem as follows:

Input: (a, b) with $a \in \mathbb{Z}$ and $b \in \mathbb{Z}^+$

Output: a^b, which we define as follows:

$$a^b = \underbrace{a \cdot a \cdot \cdots \cdot a}_{b \text{ times}} \tag{1}$$

where "$\cdot$" denotes multiplication.

This may look like a silly problem, as powering integers is a primitive in most programming languages. However, our discussion will apply to several more complicated operators "$\cdot$" as well, for which powering is not a primitive. For now, just assume our programming language has integer multiplication as a primitive but not integer powering.

Properties The following elementary property of the Powering Problem will be useful to us: $a^b \cdot a^c = a^{b+c}$. The property holds because

$$\begin{aligned} a^b \cdot a^c &\doteq \underbrace{a \cdot a \cdot \cdots \cdot a}_{b \text{ times}} \cdot \underbrace{a \cdot a \cdot \cdots \cdot a}_{c \text{ times}} \\ &= \underbrace{a \cdot a \cdot \cdots \cdot a \cdot a \cdot a \cdot \cdots \cdot a}_{b+c \text{ times}} \\ &\doteq a^{b+c} \end{aligned}$$

To be precise, we should specify for which values of a, b, c the property holds. It holds whenever (a, b) and (a, c) are valid inputs to the Powering Problem, *i.e.*, whenever $a \in \mathbb{Z}$ and $b, c \in \mathbb{Z}^+$. We can make the range of the variables explicit by quantifying the variables involved in the formula using a *universal quantifier*, denoted as $\forall$ (the flipped first letter of "All").

Proposition 1.

$$(\forall a \in \mathbb{Z}) \left(\forall b, c \in \mathbb{Z}^+\right) a^b \cdot a^c = a^{b+c} \tag{2}$$

Similarly, the following elementary property holds.

Proposition 2.

$$(\forall a \in \mathbb{Z}) \left(\forall b, c \in \mathbb{Z}^+\right) (a^b)^c = a^{b \cdot c} \tag{3}$$

The left-hand side of (3) represents the following operation: take a, raise it to the power b, and raise the result to the power c. The argument for (3) is similar to the one for (2):

Proof. Let $a \in \mathbb{Z}$ and $b, c \in \mathbb{Z}^+$ be arbitrary. We have

$$\begin{aligned} (a^b)^c &\doteq \underbrace{a^b \cdot a^b \cdot \cdots \cdot a^b}_{c \text{ times}} \\ &\doteq \underbrace{\underbrace{a \cdot a \cdot \cdots \cdot a}_{b \text{ times}} \cdot \underbrace{a \cdot a \cdot \cdots \cdot a}_{b \text{ times}} \cdot \cdots \cdot \underbrace{a \cdot a \cdot \cdots \cdot a}_{b \text{ times}}}_{c \text{ times}} \\ &= \underbrace{a \cdot a \cdot \cdots \cdot a \cdot a \cdot a \cdot \cdots \cdot a \cdot \cdots \cdot a \cdot a \cdot \cdots \cdot a}_{b \cdot c \text{ times}} \\ &\doteq a^{b \cdot c}. \end{aligned}$$

□

The restriction $a \in \mathbb{Z}$ in our specification of the Powering Problem is rather arbitrary. The formula (1) makes sense for any $a \in \mathbb{R}$, as long as $b \in \mathbb{Z}^+$. We can use it to define a^b for such a and b. The following elementary property makes use of that extension.

Proposition 3.

$$(\forall a \in \mathbb{R})\,(\forall b \in \mathbb{R} \setminus \{0\})\,(\forall c \in \mathbb{Z}^+)\,\left(\frac{a}{b}\right)^c = \frac{a^c}{b^c} \tag{4}$$

Proof. Let $a \in \mathbb{R}$, $b \in \mathbb{R} \setminus \{0\}$, and $c \in \mathbb{Z}^+$ be arbitrary. We have

$$\begin{aligned}
\left(\frac{a}{b}\right)^c &\doteq \underbrace{\frac{a}{b} \cdot \frac{a}{b} \cdot \cdots \cdot \frac{a}{b}}_{c \text{ times}} \\
&= \frac{\overbrace{a \cdot a \cdot \cdots \cdot a}^{c \text{ times}}}{\underbrace{b \cdot b \cdot \cdots \cdot b}_{c \text{ times}}} \\
&\doteq \frac{a^c}{b^c}
\end{aligned}$$

□

Exponentials and logarithms In fact, for any fixed $a \in \mathbb{R}^+$, it is possible to extend the definition of a^b to any $b \in \mathbb{Q}$ in a unique way while maintaining properties (2), (3), and (4) on all valid inputs. For example, by plugging in $b = 0$ in (2) we get that $a^0 \cdot a^c = a^c$, which implies $a^0 = 1$. Using continuity, we can further uniquely extend the definition of a^b to every $b \in \mathbb{R}$. This results in the exponential function with base a, mapping any $x \in \mathbb{R}$ to a^x. See Figure 1 for a plot with $a > 1$.

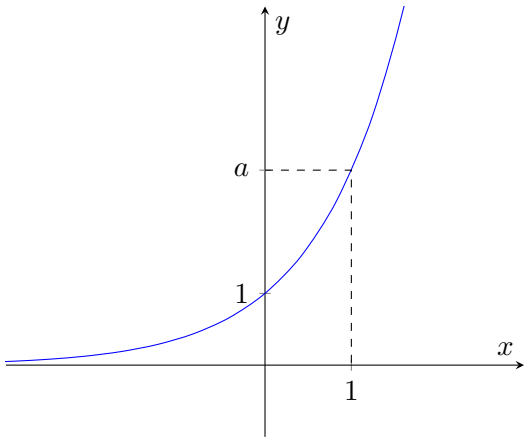

Figure 1: Plot of $y = a^x$ for $a > 1$

As the exponential function with base $a \neq 1$ is monotone (increasing for $a > 1$, and decreasing for $a < 1$) and takes on all values in $\mathbb{R}^+$, it can be inverted on $\mathbb{R}^+$, resulting in the logarithmic function with base a:

$$\log_a x = y \Leftrightarrow a^y = x$$

where $\Leftrightarrow$ denotes logical equivalence, *i.e.*, "if and only if" (which we often abbreviate to "iff"). A plot of the logarithmic function can be obtained by flipping the plot of the corresponding exponential function over the axis $x = y$. See Figure 2 for an example with $a > 1$.

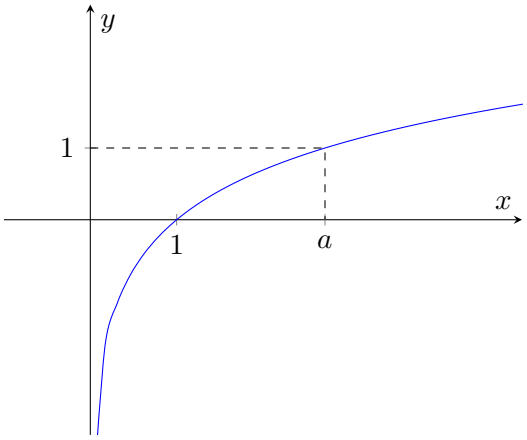

Figure 2: Plot of $y = \log_a x$ for $a > 1$

Property (2) (more precisely, the extension we discussed) translates into

$$\left(\forall a, y, z \in \mathbb{R}^+\right) \log_a(y \cdot z) = \log_a(y) + \log_a(z)$$

By default, logarithms in this course have base $a = 2$. For example, $n^{\log n} = (2^{\log n})^{\log n} = 2^{(\log n)\cdot(\log n)}$ for any $n \in \mathbb{Z}^+$.

3 An Iterative Powering Algorithm

Below is a simple iterative algorithm for the Powering Problem, assuming the multiplication operator "$\cdot$" as a primitive. It is a straightforward "algorithmization" of Equation (1).

Algorithm 1

Input: $a \in \mathbb{Z}, b \in \mathbb{Z}^+$
Output: a^b

```
1: procedure IT-POWER(a, b)
2:     r ← a
3:     c ← 1
4:     while c < b do
5:         r ← r · a
6:         c ← c + 1
7:     return r
```

In order to prove Algorithm 1 correct, we apply the two-step approach—we first establish partial correctness and then termination.

3.1 Partial correctness

We need to show that for all $a \in \mathbb{Z}$ and $b \in \mathbb{Z}^+$, if the algorithm terminates on input (a, b), then it returns a^b. Since the program returns r once it gets out of the loop, showing partial correctness for this program boils down to proving that if the loop is exited, then at that point in time $r = a^b$.

Loop invariants We will do so by establishing an appropriate loop invariant. A *loop invariant* is a statement that holds each time the loop condition is about to be tested. For the loop in Algorithm 1, this is right before the condition "$c < b$" in Line (4) is tested.

Claim 1. *The following are loop invariants for Algorithm 1:*

$$r = a^c \tag{5}$$
$$c \in \mathbb{Z} \tag{6}$$
$$c \leq b \tag{7}$$

Before proving Claim 1, let us see how we can use it to establish partial correctness. Suppose that on a given valid input (a, b), the loop is exited, and consider the point in time right before the condition "$c < b$" is tested of the loop iteration that is exited. We know that

- All of (5), (6), and (7) hold (by the invariants stated in Claim 1), and
- The negation of the loop condition holds, *i.e.*, $\neg\,(c < b)$, where $\neg$ denotes negation (since the loop is exited)

Since $\neg\,(c < b)$ is equivalent to $c \geq b$ and (7) states that $c \leq b$, we have that $c = b$. Combined with (5), this means that $r = a^b$, which is what we needed to show.

Mathematical induction What remains is to establish the loop invariants, *i.e.*, to prove Claim 1. In general, loop invariants are statements of the form

$$\left(\forall t \in \mathbb{Z}^+\right) P(t) \tag{8}$$

where $P(t)$ expresses that if the loop condition is about to be tested for the tth time, then the invariant holds at that point in time. An approach to establish statements of the form (8) that is particularly suited for loop invariants is *induction*. The approach consists of two steps:

Base case Show that $P(1)$ holds.

Induction step Show that $(\forall t \in \mathbb{Z}^+)\, P(t) \Rightarrow P(t+1)$ holds, where $\Rightarrow$ denotes logical implication. In words: Show that for every positive integer t, if $P(t)$ holds, then $P(t+1)$ holds. The premise $P(t)$ in this step is called the induction hypothesis.

Those two steps combined establish (8): We know that $P(1)$ holds by the base case. The base case combined with the induction step for $t = 1$ implies that $P(2)$ holds. The fact that $P(2)$ holds combined with the induction step for $t = 2$ implies that $P(3)$ holds, and so on.

To illustrate the principle of mathematical induction, we include a proof by induction of the following well-known formula for the sum of the first n positive integers, $\sum_{i=1}^{n} i \doteq 1+2+3+\cdots+n$.

Proposition 4. $(\forall n \in \mathbb{Z}^+) \sum_{i=1}^{n} i = \frac{n(n+1)}{2}$

Proof. The statement we need to prove is of the form (8) (using n instead of t as the universally quantified variable), where $P(n)$ denotes the statement $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$. We prove the statement by induction on n.

Base case: $P(1)$ states that $\sum_{i=1}^{1} i = 1$, which trivially holds.

Induction step: Let $n \in \mathbb{Z}^+$ be arbitrary, and assume that $P(n)$ holds. We have that

$$\sum_{i=1}^{n+1} i = \left(\sum_{i=1}^{n} i\right) + (n+1) = \frac{n(n+1)}{2} + (n+1) = \frac{(n+1)(n+2)}{2}$$

where the middle equality uses the induction hypothesis $P(n)$. The resulting equality $\sum_{i=1}^{n+1} i = \frac{(n+1)(n+2)}{2}$ is exactly $P(n+1)$.

□

A closely related proof strategy is *strong induction*, in which the induction step is modified as follows:

$$\left(\forall t \in \mathbb{Z}^+\right) P(1) \wedge P(2) \wedge \cdots \wedge P(t) \Rightarrow P(t+1)$$

Sometimes the smallest value in the range for t is an integer b other than 1; the base case changes accordingly to establishing $P(b)$. Sometimes there are multiple base cases, and the smallest value of the range for t in the induction step is larger than b.

Establishing loop invariants by induction Loop invariants can be established by induction on the loop iteration. We now do so for Algorithm 1.

Proof of Claim 1. Fix any valid input (a, b).

Base case We need to show that (5), (6), and (7) hold when the loop condition "$c < b$" is tested for the first time. At that point, $r = a$ and $c = 1$ by the assignments that were just executed. Thus $r = a^c$, $c \in \mathbb{Z}^+$, and $c \leq b$ as any valid input (a, b) satisfies $b \geq 1$.

Induction step For every $t \in \mathbb{Z}^+$ we need to show that, if the loop condition "$c < b$" is tested for the $(t+1)$st time, and all invariants (5), (6), and (7) hold the tth time the loop condition "$c < b$" is tested, then they also hold the $(t+1)$st time the loop condition is tested.

In order to argue this, we introduce notation to reference the values of variables that change over the duration of the program. Let us denote by r_t the value of r right before the tth time the loop condition is tested. We define c_t similarly.

Suppose that there is a $(t+1)$st time that the loop condition "$c < b$" is tested. As this implies that there was a tth time that the condition was tested, by the induction hypothesis, we know that $r_t = a^{c_t}$ (5), $c_t \in \mathbb{Z}^+$ (6), and $c_t \leq b$ (7). Moreover, the loop condition "$c < b$" held during the tth iteration, so $c_t < b$, and that iteration set $r_{t+1} = r_t \cdot a$ and $c_{t+1} = c_t + 1$. All together, we have that

$$r_{t+1} = r_t \cdot a = a^{c_t} \cdot a = a^{c_t+1} = a^{c_{t+1}}$$

establishing (5) for the $(t+1)$st iteration. Since $c_t \in \mathbb{Z}^+$ and $c_{t+1} = c_t + 1$, it follows that $c_{t+1} \in \mathbb{Z}^+$, establishing (6) for the $(t+1)$st iteration. Since c_t and b are integers, $c_t < b$ implies $c_t \leq b - 1$, so c_{t+1} satisfies

$$c_{t+1} = c_t + 1 \leq (b-1) + 1 = b$$

which establishes (7) for the $(t+1)$st iteration. This finishes the induction step.

□

Setting up loop invariants Once appropriate loop invariants are found, establishing them by induction is usually straightforward. Coming up with appropriate loop invariants typically requires more ingenuity. Relationships between variables that express their meaning in a precise way are often helpful to include. Ultimately, the loop invariants need to be strong enough to enable the partial correctness proof.

Additional invariants are sometimes included for the sake of the termination argument, or to facilitate the induction proof that the invariants hold. For example, only invariants (5) and (7) were needed to establish partial correctness for Algorithm 1, but (6) was needed in the induction step of the proof that (5) and (7) are invariants.

Loop invariants can also be helpful in the *design* of an algorithm. If one first comes up with the right loop invariants, they can be used to figure out how the loop variables should be initialized, how they should be updated in the body of the loop, and what the loop condition should be.

3.2 Termination

In order to establish termination of Algorithm 1, we need to show that the algorithm halts on every input (a, b) with $a \in \mathbb{Z}$ and $b \in \mathbb{Z}^+$. The only reason the algorithm may not halt is because it gets stuck in an infinite loop. Since the loop is exited as soon as $c \geq b$ and c increases by 1 in each iteration of the loop, the loop cannot be infinite.

4 A Recursive Powering Algorithm

Recursive procedures can make calls to themselves. The loop in Algorithm 1 can be avoided by rewriting the algorithm in recursive form as in Algorithm 2 below.

Algorithm 2

Input: $a \in \mathbb{Z}, b \in \mathbb{Z}^+$
Output: a^b

```
1: procedure REC-POWER(a, b)
2:     if b = 1 then
3:         return a
4:     else
5:         return REC-POWER(a, b − 1) · a
```

Algorithm 2 executes the same multiplications as our iterative algorithm. However, the code of Algorithm 2 is more succinct. The use of recursion obviates the need for a loop and the corre-

sponding loop variables r and c. Instead, the compiler needs to do more work to keep track of the progress and store intermediate results.

Recursion tree In order to analyze a recursive algorithm A, it often helps to think in terms of the recursion tree A induces. The *recursion tree* of A on a given input x is a rooted tree that contains a node for every call to A that is made during the execution of A on input x. The root corresponds to the original call to A with argument x. The children of a call with argument y correspond to the calls to A that appear in the code for A on input y. The leaves of the tree correspond to the base cases of the recursion, *i.e.*, arguments y for which A returns the result without making any recursive calls.

For Algorithm 2 on input $(a, b) = (10, 3)$, the recursion tree is depicted in Figure 3. It consists of a line with three nodes, namely corresponding to the arguments $(10, 3)$, $(10, 2)$, and $(10, 1)$. The nodes of the tree are labeled with the arguments of the call they represent. As usual, we draw the tree upside down—with the root on top.

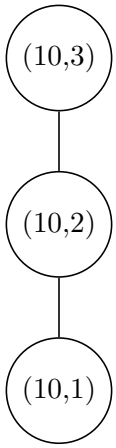

Figure 3: Recursion tree of Algorithm 2 on input $(a, b) = (10, 3)$

Arguing correctness of a recursive procedure can be established in the usual two steps: partial correctness and termination. Alternately, correctness can be established in one shot by structural induction. We explain both approaches and illustrate them for Algorithm 2.

4.1 Partial correctness

Partial correctness of a recursive procedure follows if for every valid input x:

- All the recursive calls that appear in the code of the program on input x have valid arguments.
- Assuming all those calls return a correct output for their respective arguments, and that the program terminates on input x, the program returns a correct output on input x.

Note that the first bullet (taken over all valid inputs) implies that all the nodes in the recursion tree on a valid input have valid arguments. The second bullet then implies that if the procedure terminates on a given valid input, it returns a correct output.

Algorithm 2 satisfies partial correctness because for every $a \in \mathbb{Z}$ and $b \in \mathbb{Z}^+$:

- The only recursive call that appears in the code of the program is the one in Line (5), with arguments $(a, b-1)$. That call is only made when $b \neq 1$, in which case $b-1 \in \mathbb{Z}^+$. It follows that all the calls that appear in the code of the program on a valid input have valid arguments.

- The only places where the program returns an answer are in Lines 3 and 5. Line 3 is only executed if $b = 1$, in which case a is the correct output. Assuming the recursive call with arguments $(a, b-1)$ in Line 5 returns the correct answer, namely a^{b-1}, the program returns $a^{b-1} \cdot a = a^b$, which is correct.

4.2 Termination

Termination boils down to showing that the recursion bottoms out on every valid input, *i.e.*, every branch in the recursion tree eventually reaches a base case. In general, this can be proved by defining a relation $\prec$ on valid inputs with the following properties:

- $y \prec x$ whenever x is a valid input and y is the argument of a recursive call that appears in the code of the program on input x.

- There does not exist an infinite downward chain of valid inputs, i.e., an infinite sequence of valid inputs $y_1, y_2, \ldots$ such that $y_1 \succ y_2 \succ \ldots$.

The existence of such a relation implies termination as all arguments in recursive calls on a valid input can be assumed valid by the first part of the partial correctness argument.

Often times the relation $\prec$ is induced by a potential function μ that associates every valid input x to a non-negative integer $\mu(x)$ such that $\mu(y) < \mu(x)$ whenever x is a valid input and y is the argument of a recursive call that appears in the code of the program on input x. Defining $y \prec x$ iff $\mu(y) < \mu(x)$ satisfies both properties required of $\prec$. In fact, the properties of the potential function imply that the depth of the recursion tree for a valid input x cannot exceed $\mu(x)$. The value $\mu(x)$ can be viewed as a measure of the complexity of x with respect to the recursion, and is often closely related to the depth of the recursion tree.

In our example, we can simply set $\mu(a, b) = b$. The key points are that b is a positive integer for any valid input (a, b), and that $\mu(a, b-1) = b - 1 = \mu(a, b) - 1 < \mu(a, b)$. Note that the depth of the recursion tree on input (a, b) equals $b - 1$.

4.3 Correctness in one shot

Note that the partial correctness and termination proofs are more intertwined than for the iterative algorithm, as the termination proof assumes that all recursive calls have valid arguments.

Alternately, given a potential function μ as before, one can establish correctness in one shot by strong induction on the value of μ. That is, one shows by induction on m that

$$(\forall m \in \{-1\} \cup \mathbb{N})\, P(m),$$

where $P(m)$ expresses that the procedure returns a correct output on all valid inputs x with $\mu(x) \leq m$. The base case $P(-1)$ holds vacuously, as there are no valid inputs with $\mu \leq -1$. In the induction step, the fact that $\mu(y) < \mu(x)$ for a recursive call with a valid argument y made on a valid input x, implies by the induction hypothesis that the recursive call returns a correct answer.

This type of inductive argument is referred to as *structural induction*.

In our example with potential function $\mu(a, b) = b$, we argue the induction step $P(m) \Rightarrow P(m+1)$ for $m \in \mathbb{N}$ as follows: Consider any (a, b) with $a \in \mathbb{Z}$, $b \in \mathbb{Z}^+$, and $b \leq m+1$. If the test in Line 2 is passed, then $b = 1$ and the procedure correctly returns a. Otherwise, $b > 1$ and the procedure executes Line 5. The argument $(a, b-1)$ to the recursive call is valid (as $b - 1 > 0$) and has potential $\mu(a, b-1) = b - 1 \leq (m+1) - 1 = m$. Thus, by the induction hypothesis the recursive call returns a^{b-1}, from which it follows that the procedure correctly returns a^b.

5 Greatest Common Divisor

The second problem we use to illustrate program correctness also deals with integers, namely the Greatest Common Divisor Problem, or GCD Problem for short.

Definitions We first review some definitions related to integer divisibility. For positive integers a and b, we say that a divides b if b can be broken into whole copies of a, *i.e.*, b can be written in the form $b = a + a + \cdots + a$. More generally, for arbitrary integers a and b, we say that a divides b, or a is a *divisor* of b, if b can be written in the form $b = a \cdot c$ for some integer c. In symbols:

$$a \mid b \Leftrightarrow (\exists c \in \mathbb{Z})\ b = a \cdot c$$

where the symbol $\exists$ (the flipped first letter of "Exists") denotes an *existential quantifier*, and "$(\exists c \in \mathbb{Z})$" reads "there exists an element c in $\mathbb{Z}$ such that".

Every integer a has at least 1, a, -1, and $-a$ as divisors, but may have more. For example, the set of divisors of 6 is $\{-6, -3, -2, -1, 1, 2, 3, 6\}$. The set of divisors of 0 is $\mathbb{Z}$. For every integer other than 0 the set of divisors is finite.

The *common divisors* of two integers a and b are the integers that divide both a and b. The *greatest common divisor* of a and b, denoted $\gcd(a, b)$, is the largest element in the set of common divisors (if that element exists).

For example, $\gcd(6, 6) = 6$, $\gcd(5, 6) = 1$, $\gcd(4, 6) = 2$, $\gcd(0, 6) = 6$, and $\gcd(0, 0)$ is undefined as the set of common divisors of 0 and 0 is $\mathbb{Z}$, which does not have a largest element. For every pair (a, b) of integers other than $(0, 0)$, $\gcd(a, b)$ is well-defined, as the set of common divisors is non-empty and finite.

Properties We now develop some properties of greatest common divisors that are helpful to compute them.

Claim 2. *For any two integers a and b, the set of common divisors of a and b is the same as the set of common divisors of a and $b - a$.*

Proof. Consider an integer d that divides both a and b. By definition, we can write $a = d \cdot e$ and $b = d \cdot f$ for some integers e and f. It follows that $b - a = d \cdot (f - e)$. Since $f - e$ is an integer, this means that d divides $b - a$.

Thus, every common divisor of a and b is also a common divisor of a and $b - a$. The opposite inclusion follows similarly, completing the proof of the claim. □

The claim implies the following invariance property for the greatest common divisor:

Proposition 5.

$$(\forall a, b \in \mathbb{Z})\, \gcd(a, b) = \gcd(a, b - a) = \gcd(a - b, b) \tag{9}$$

where "=" means that either both sides of the equation are well-defined and equal, or else both sides are undefined.

Proof. The first equality follows immediately from the claim. The second equality follows from the first one and the fact that gcd is symmetric in its arguments:

$$\gcd(a, b) = \gcd(b, a) = \gcd(b, a - b) = \gcd(a - b, b)$$

□

Another useful property is the following generalization of our example that $\gcd(6, 6) = 6$.

Proposition 6.

$$\left(\forall a \in \mathbb{Z}^+\right) \gcd(a, a) = a \tag{10}$$

Proof. Any $a \in \mathbb{Z}^+$ has itself as its largest divisor. □

Specification Even though $\gcd(a, b)$ is well-defined for every pair (a, b) of integers other than $(0, 0)$, we will keep things simple by imposing the precondition that both a and b are positive. Thus, we specify the Greatest Common Divisor Problem as follows:

Input: (a, b) with $a, b \in \mathbb{Z}^+$

Output: $\gcd(a, b)$

6 An Iterative GCD Algorithm

The idea behind our first algorithm for the GCD Problem is to iteratively use Proposition 5 to bring the arguments closer to each other without affecting the gcd. Eventually, the arguments will coincide, in which case we know the answer by virtue of Proposition 6. Put another way, as long as the inputs do not coincide, we subtract the smaller one from the larger one. This leads to Algorithm 3.

We show correctness by arguing partial correctness and termination.

Partial correctness We argue partial correctness via loop invariants.

Claim 3. *The following are loop invariants for Algorithm 3:*

$$x, y \in \mathbb{Z}^+$$
$$\gcd(x, y) = \gcd(a, b)$$

Proof. Fix a valid input $a, b \in \mathbb{Z}^+$. The invariants hold initially because (x, y) is initialized to (a, b). The first invariant is maintained between iterations of the while loop because

- If y is modified in the while loop, it is in Line 6 by subtracting x. Since x is an integer that is strictly less than y (as the test in Line 5 was passed), we can conclude that we still have $y \in \mathbb{Z}^+$.

Algorithm 3

Input: $a, b \in \mathbb{Z}^+$
Output: $\gcd(a, b)$

```
1: procedure IT-GCD(a, b)
2:     x ← a
3:     y ← b
4:     while x ≠ y do
5:         if x < y then
6:             y ← y − x
7:         else
8:             x ← x − y
9:     return x
```

- If x is modified in the while loop, it is Line 8 by subtracting an integer y. Since y is an integer that is strictly less than x (as the test in Line 4 was passed and the test in Line 5 failed), we can conclude that we still have $x \in \mathbb{Z}^+$.

The second invariant is maintained because Proposition 5 implies that none of the operations in the body of the loop affects $\gcd(x, y)$. □

Partial correctness follows because, if the program terminates on input (a, b), then the while loop has been exited, which means the test condition in Line 4 fails to hold, and therefore $x = y$. Since $x \in \mathbb{Z}^+$ by the second invariant, Proposition 6 tells us that $\gcd(x, y) = x$. Furthermore, $\gcd(a, b) = \gcd(x, y)$ by the first invariant. All together we have that $\gcd(a, b) = x$, and returning x in Line 9 is correct.

Termination In each iteration of the loop, exactly one of x or y is decreased by the other one, while the other one remains unchanged. As this "other one" has a positive integer value (by the first invariant), each iteration decreases $x + y$ by a positive integer amount. Since $x + y$ remains positive itself (again by the first invariant), this means that the loop must terminate.

7 A Recursive GCD Algorithm

Algorithm 3 can be rewritten in recursive form as in Algorithm 4. Algorithm 4 is *tail-recursive*, *i.e.*, the results of recursive calls are propagated up the recursion tree without any further operation.

Correctness We argue correctness by structural induction. We use the potential function $\mu(a, b) \doteq a+b$, which only takes on positive integer values for $a, b \in \mathbb{Z}^+$. We prove that $(\forall m \in \mathbb{N})\, P(m)$, where $P(m)$ states that Algorithm 4 correctly returns $\gcd(a, b)$ on all inputs $a, b \in \mathbb{Z}^+$ with $a + b \leq m$.

The base case $m = 0$ holds vacuously; there are no positive integers a and b for which $a+b = 0$. For the induction step, $P(m) \Rightarrow P(m + 1)$ with $m \in \mathbb{N}$, let $a, b \in \mathbb{Z}^+$ with $a + b \leq m + 1$. If $a = b$ then Line 3 is executed and the procedure returns a, which is correct by Proposition 6. Otherwise, Line 6 is executed if $a < b$, and Line 8 if $a > b$. In either case, a recursive call is made, say with argument (x, y), where $x, y \in \mathbb{Z}^+$. The sum $x+y$ equals b or a, which are integers strictly less than $a+b \leq m+1$. Thus, $x+y \leq m$, and the induction hypothesis $P(m)$ applies to the recursive call, *i.e.*,

Algorithm 4

Input: $a, b \in \mathbb{Z}^+$
Output: $\gcd(a, b)$

```
1: procedure REC-GCD(a, b)
2:     if a = b then
3:         return a
4:     else
5:         if a < b then
6:             return REC-GCD(a, b − a)
7:         else
8:             return REC-GCD(a − b, b)
```

the recursive call correctly returns $\gcd(x, y)$. By Proposition 5, in both cases $\gcd(x, y) = \gcd(a, b)$, so the procedure on input (a, b) correctly returns $\gcd(a, b)$.

Euclid's algorithm If $a < b$, Algorithm 4 keeps subtracting a from the second argument until the latter no longer exceeds a. We can do all of those subtractions at once by replacing the second argument by

$$\begin{array}{ll} a & \text{if } a \mid b \\ b \bmod a & \text{otherwise} \end{array}$$

where $b \bmod a$ denotes the remainder of the division of b by a. In fact, we can apply the second replacement in both cases, provided we allow one of the arguments to the procedure to become 0, and return the other argument whenever that happens. Also, since $b \bmod a < a$, we know the way the inequality goes between the new argument, so there is no need to test it.

This leads to the slick implementation given in Algorithm 5. Note the relaxed precondition in the specification (allowing one of the arguments to be 0), and the use of an auxiliary procedure with the precondition that the arguments are in order.

Algorithm 5 and its iterative counterpart are known as Euclid's algorithm. We leave their correctness as an exercise.

Algorithm 5

Input: $a, b \in \mathbb{N}$ with $(a, b) \neq (0, 0)$
Output: $\gcd(a, b)$

```
1: procedure EUCLID(a, b)
2:     if a ≤ b then
3:         return REC-EUCLID(a, b)
4:     else
5:         return REC-EUCLID(b, a)
```

Input: $a, b \in \mathbb{N}$ with $(a, b) \neq (0, 0)$ and $a \leq b$
Output: $\gcd(a, b)$

```
6: procedure REC-EUCLID(a, b)
7:     if a = 0 then
8:         return b
9:     else
10:        return REC-EUCLID(b mod a, a)
```