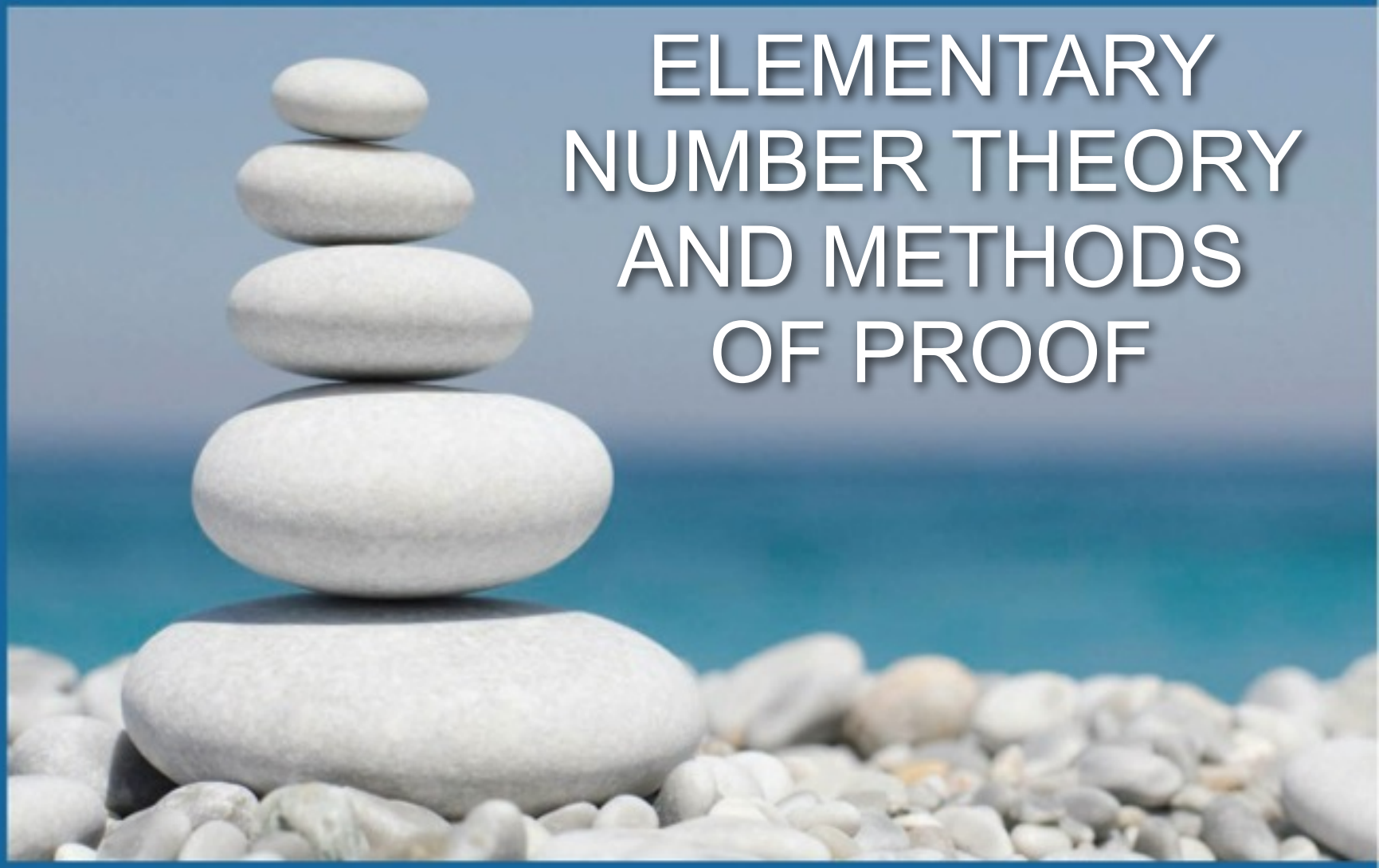
CHAPTER 4



ELEMENTARY NUMBER THEORY AND METHODS OF PROOF

SECTION 4.8

Application: Algorithms

Application: Algorithms

The word *algorithm* refers to a step-by-step method for performing some action.

Some examples of algorithms in everyday life are food preparation recipes, directions for assembling equipment or hobby kits, sewing pattern instructions, and instructions for filling out income tax forms.

Much of elementary school mathematics is devoted to learning algorithms for doing arithmetic such as multidigit addition and subtraction, multidigit (or long) multiplication, and long division.

An Algorithmic Language

An Algorithmic Language

The algorithmic language used in this book is a kind of pseudocode, combining elements of Pascal, C, Java, and VB.NET, and ordinary, but fairly precise, English.

We will use some of the formal constructs of computer languages—such as assignment statements, loops, and so forth—but we will ignore the more technical details, such as the requirement for explicit end-of-statement delimiters, the range of integer values available on a particular installation, and so forth.

An Algorithmic Language

The algorithms presented in this text are intended to be precise enough to be easily translated into virtually any high-level computer language.

In high-level computer languages, the term **variable** is used to refer to a specific storage location in a computer's memory.

To say that the variable *x* has the value 3 means that the memory location corresponding to *x* contains the number 3.

An Algorithmic Language

A given storage location can hold only one value at a time. So if a variable is given a new value during program execution, then the old value is erased.

The **data type** of a variable indicates the set in which the variable takes its values, whether the set of integers, or real numbers, or character strings, or the set {0, 1} (for a Boolean variable), and so forth.

An Algorithmic Language

An **assignment statement** gives a value to a variable. It has the form

$$x := e,$$

where *x* is a variable and *e* is an expression. This is read "*x* is assigned the value *e*" or "let *x* be *e*."

An Algorithmic Language

When an assignment statement is executed, the expression *e* is evaluated (using the current values of all the variables in the expression), and then its value is placed in the memory location corresponding to *x* (replacing any previous contents of this location).

Ordinarily, algorithm statements are executed one after another in the order in which they are written.

An Algorithmic Language

Conditional statements allow this natural order to be overridden by using the current values of program variables to determine which algorithm statement will be executed next.

Conditional statements are denoted either

a. **if** (*condition*)
then s_1
else s_2

or

b. **if** (*condition*) **then** s_1

where *condition* is a predicate involving algorithm variables and where s_1 and s_2 are algorithm statements or groups of algorithm statements.

An Algorithmic Language

We generally use indentation to indicate that statements belong together as a unit.

When ambiguity is possible, however, we may explicitly bind a group of statements together into a unit by preceding the group with the word **do** and following it with the words **end do.**

An Algorithmic Language

Execution of an **if-then-else** statement occurs as follows:

1. The *condition* is evaluated by substituting the current values of all algorithm variables appearing in it and evaluating the truth or falsity of the resulting statement.

2. If *condition* is true, then s_1 is executed and execution moves to the next algorithm statement following the **if-then-else** statement.

3. If *condition* is false, then s_2 is executed and execution moves to the next algorithm statement following the **if-then-else** statement.

An Algorithmic Language

Execution of an **if-then** statement is similar to execution of an **if-then-else** statement, except that *if condition* is false, execution passes immediately to the next algorithm statement following the **if-then** statement.

Often *condition* is called a **guard** because it is stationed before s_1 and s_2 and restricts access to them.

Example 1 – *Execution of if-then-else and if-then Statements*

Consider the following algorithm segments:

a. **if** $x > 2$
then $y := x + 1$
else do $x := x - 1$
$y := 3 \cdot x$ **end do**

b. $y := 0$
if $x > 2$ **then** $y := 2^x$

What is the value of *y* after execution of these segments for the following values of *x*?

i. $x = 5$

ii. $x = 2$

Example 1(a) – *Solution*

(i) Because the value of x is 5 before execution, the guard condition $x > 2$ is true at the time it is evaluated. Hence the statement following **then** is executed, and so the value of $x + 1 = 5 + 1$ is computed and placed in the storage location corresponding to y.

So after execution, $y = 6$.

(ii) Because the value of x is 2 before execution, the guard condition $x > 2$ is false at the time it is evaluated.

Hence the statement following **else** is executed.

Example 1(a) – *Solution*

cont'd

The value of $x - 1 = 2 - 1$ is computed and placed in the storage location corresponding to x, and the value of $3 \bullet x = 3 \bullet 1$ is computed and placed in the storage location corresponding to y. So after execution, $y = 3$.

Example 1(b) – *Solution*

cont'd

(i) Since $x = 5$ initially, the condition $x > 2$ is true at the time it is evaluated. So the statement following **then** is executed, and y obtains the value $2^5 = 32$.

(ii) Since $x = 2$ initially, the condition $x > 2$ is false at the time it is evaluated. Execution, therefore, moves to the next statement following the if-then statement, and the value of y does not change from its initial value of 0.

An Algorithmic Language

Iterative statements are used when a sequence of algorithm statements is to be executed over and over again. We will use two types of iterative statements: **while** loops and **for-next** loops.

A **while** loop has the form

while (*condition*)

[statements that make up
the body of the loop]

end while

where *condition* is a predicate involving algorithm variables.

An Algorithmic Language

The word **while** marks the beginning of the loop, and the words **end while** mark its end.

Execution of a **while** loop occurs as follows:

1. The *condition* is evaluated by substituting the current values of all the algorithm variables and evaluating the truth or falsity of the resulting statement.

2. If *condition* is true, all the statements in the body of the loop are executed in order. Then execution moves back to the beginning of the loop and the process repeats.

An Algorithmic Language

3. If *condition* is false, execution passes to the next algorithm statement following the loop.

The loop is said to be **iterated** (IT-a-rate-ed) each time the statements in the body of the loop are executed.

Each execution of the body of the loop is called an **iteration** (it-er-AY-shun) of the loop.

Example 2 – *Tracing Execution of a **while** Loop*

Trace the execution of the following algorithm segment by finding the values of all the algorithm variables each time they are changed during execution:

$i := 1, s := 0$

while $(i \leq 2)$

$\quad s := s + i$

$\quad i := i + 1$

end while

Example 2 – *Solution*

Since i is given an initial value of 1, the condition $i \leq 2$ is true when the **while** loop is entered.

So the statements within the loop are executed in order:

$$s = 0 + 1 = 1 \quad \text{and} \quad i = 1 + 1 = 2.$$

Then execution passes back to the beginning of the loop.

Example 2 – *Solution*

cont'd

The condition $i \leq 2$ is evaluated using the current value of i, which is 2.

The condition is true, and so the statements within the loop are executed again:

$$s = 1 + 2 = 3 \quad \text{and} \quad i = 2 + 1 = 3.$$

Then execution passes back to the beginning of the loop.

Example 2 – *Solution*

The condition $i \leq 2$ is evaluated using the current value of i, which is 3. This time the condition is false, and so execution passes beyond the loop to the next statement of the algorithm.

This discussion can be summarized in a table, called a **trace table**, that shows the current values of algorithm variables at various points during execution.

Variable Name	Iteration Number 0	1	2
i	1	2	3
s	0	1	3

Trace Table

Example 2 – *Solution*

cont'd

The trace table for a **while** loop generally gives all values immediately following each iteration of the loop.
("After the zeroth iteration" means the same as "before the first iteration.")

An Algorithmic Language

The second form of iteration we will use is a **for-next** loop. A **for-next** loop has the following form:

for *variable* := *initial expression* **to** *final expression*

[statements that make up
the body of the loop]

next *(same) variable*

A **for-next** loop is executed as follows:

1. The **for-next** loop variable is set equal to the value of *initial expression*.

An Algorithmic Language

2. A check is made to determine whether the value of *variable* is less than or equal to the value of *final expression*.

3. If the value of *variable* is less than or equal to the value of *final expression*, then the statements in the body of the loop are executed in order, *variable* is increased by 1, and execution returns back to step 2.

4. If the value of *variable* is greater than the value of *final expression*, then execution passes to the next algorithm statement following the loop.

Example 3 – *Trace Table for a **for-next** Loop*

Convert the **for-next** loop shown below into a **while** loop. Construct a trace table for the loop.

for $i := 1$ **to** 4

$x := i^2$

next i

Example 3 – *Solution*

The given **for-next** loop is equivalent to the following:

$i := 1$

while $(i \leq 4)$

$x := i^2$

$i := i + 1$

end while

Its trace table is as follows:

Variable Name	Iteration Number 0	1	2	3	4
x		1	4	9	16
i	1	2	3	4	5

Trace Table

A Notation for Algorithms

A Notation for Algorithms

We generally include the following information when describing algorithms formally:

1. The name of the algorithm, together with a list of input and output variables.

2. A brief description of how the algorithm works.

3. The input variable names, labeled by data type (whether integer, real number, and so forth).

A Notation for Algorithms

4. The statements that make up the body of the algorithm, possibly with explanatory comments.

5. The output variable names, labeled by data type.

The Division Algorithm

The Division Algorithm

For an integer *a* and a positive integer *d*, the quotient-remainder theorem guarantees the existence of integers *q* and *r* such that

$$a = dq + r \quad \text{and} \quad 0 \leq r < d.$$

In this section, we give an algorithm to calculate *q* and *r* for given *a* and *d* where a is nonnegative.

The Division Algorithm

Algorithm 4.8.1 Division Algorithm :
[Given a nonnegative integer a and a positive integer d, the aim of the algorithm is to find integers q and r that satisfy the conditions $a = dq + r$ *and* $0 \leq r < d$.

This is done by subtracting d repeatedly from a until the result is less than d but is still nonnegative.

$$0 \leq a - d - d - d - \cdots - d = a - dq < d.$$

The total number of d's that are subtracted is the quotient q. The quantity a – dq equals the remainder r.]

The Division Algorithm

Input: *a [a nonnegative integer], d [a positive integer]*

Algorithm Body:
r := *a*, *q* := 0
[Repeatedly subtract d from r until a number less than d is obtained. Add 1 to q each time d is subtracted.]

while $(r \geq d)$
$r := r - d$
$q := q + 1$
end while

[After execution of the **while** *loop, a = dq + r.]*

The Division Algorithm

Output:
q, *r [nonnegative integers]*

Note that the values of *q* and *r* obtained from the division algorithm are the same as those computed by the *div* and *mod* functions built into a number of computer languages.

That is, if *q* and *r* are the quotient and remainder obtained from the division algorithm with input *a* and *d*, then the output variables *q* and *r* satisfy

$$q = a \; div \; d \quad \text{and} \quad r = a \; mod \; d.$$

The Euclidean Algorithm

The Euclidean Algorithm

The *greatest* common divisor of two integers *a* and *b* is the largest integer that divides both *a* and *b*. For example, the greatest common divisor of 12 and 30 is 6.

The Euclidean algorithm provides a very efficient way to compute the greatest common divisor of two integers.

• Definition

Let a and b be integers that are not both zero. The **greatest common divisor** of a and b, denoted $\mathbf{gcd}(a, b)$, is that integer d with the following properties:

1. d is a common divisor of both a and b. In other words,

$$d \mid a \quad \text{and} \quad d \mid b.$$

2. For all integers c, if c is a common divisor of both a and b, then c is less than or equal to d. In other words,

$$\text{for all integers } c, \text{ if } c \mid a \text{ and } c \mid b, \text{ then } c \leq d.$$

Example 5 – *Calculating Some gcd's*

a. Find gcd(72, 63).

b. Find gcd(10^{20}, 6^{30}).

c. In the definition of greatest common divisor, gcd(0, 0) is not allowed. Why not? What would gcd(0, 0) equal if it were found in the same way as the greatest common divisors for other pairs of numbers?

Solution:

a. $72 = 9 \bullet 8$ and $63 = 9 \bullet 7$. So $9 \mid 72$ and $9 \mid 63$, and no integer larger than 9 divides both 72 and 63.

Hence *gcd*(72, 63) = 9.

Example 5 – *Solution*

cont'd

b. By the laws of exponents, $10^{20} = 2^{20} \cdot 5^{20}$ and $6^{30} = 2^{30} \cdot 3^{30} = 2^{20} \cdot 2^{10} \cdot 3^{30}$. It follows that

$$2^{20} \mid 10^{20} \quad \text{and} \quad 2^{20} \mid 6^{30},$$

and by the unique factorization of integers theorem, no integer larger than 2^{20} divides both 10^{20} and 6^{30} (because no more than twenty 2's divide 10^{20}, no 3's divide 10^{20}, and no 5's divide 6^{30}).

Hence $\gcd(10^{20}, 6^{30}) = 2^{20}$.

Example 5 – *Solution*

cont'd

c. Suppose gcd(0, 0) were defined to be the largest common factor that divides 0 and 0.

The problem is that every positive integer divides 0 and there is no largest integer.

So there is no largest common divisor!

The Euclidean Algorithm

Calculating gcd's using the approach illustrated in Example 5 works only when the numbers can be factored completely.

By the unique factorization of integers theorem, all numbers can, in principle, be factored completely. But, in practice, even using the highest-speed computers, the process is unfeasibly long for very large integers.

Over 2,000 years ago, Euclid devised a method for finding greatest common divisors that is easy to use and is much more efficient than either factoring the numbers or repeatedly testing both numbers for divisibility by successively larger integers.

The Euclidean Algorithm

The Euclidean algorithm is based on the following two facts, which are stated as lemmas.

Lemma 4.8.1

If r is a positive integer, then $\gcd(r, 0) = r$.

Lemma 4.8.2

If a and b are any integers not both zero, and if q and r are any integers such that

$$a = bq + r,$$

then

$$\gcd(a, b) = \gcd(b, r).$$

The Euclidean Algorithm

The Euclidean algorithm can be described as follows:

1. Let A and B be integers with $A > B \geq 0$.

2. To find the greatest common divisor of A and B, first check whether $B = 0$. If it is, then $\gcd(A, B) = A$ by Lemma 4.8.1.

   If it isn't, then $B > 0$ and the quotient-remainder theorem can be used to divide A by B to obtain a quotient q and a remainder r :

$$A = Bq + r \quad \text{where } 0 \leq r < B.$$

The Euclidean Algorithm

By Lemma 4.8.2, gcd(A, B) = gcd(B, r). Thus the problem of finding the greatest common divisor of A and B is reduced to the problem of finding the greatest common divisor of B and r.

What makes this piece of information useful is that B and r are smaller numbers than A and B.

To see this, recall that we assumed

$$A > B \geq 0.$$

The Euclidean Algorithm

Also the r found by the quotient-remainder theorem satisfies

$$0 \leq r < B.$$

Putting these two inequalities together gives

$$0 \leq r < B < A.$$

So the larger number of the pair (B, r) is smaller than the larger number of the pair (A, B).

3. Now just repeat the process, starting again at (2), but use B instead of A and r instead of B. The repetitions are guaranteed to terminate eventually with $r = 0$ because each new remainder is less than the preceding one and all are nonnegative.

Example 6 – *Hand-Calculation of gcd's Using the Euclidean Algorithm*

Use the Euclidean algorithm to find gcd(330, 156).

Solution:

1. Divide 330 by 156:

$$\begin{array}{r} 2 \leftarrow \text{quotient} \\ 156\overline{)\,330} \\ \underline{312} \\ 18 \leftarrow \text{remainder} \end{array}$$

Thus 330 = 156 • 2 + 18 and hence
gcd(330, 156) = gcd(156, 18) by Lemma 4.8.2.

Example 6 – *Solution*

2. Divide 156 by 18:

$$\begin{array}{r} 8 \\ 18\overline{)\,156} \\ \underline{144} \\ 12 \end{array} \quad \begin{array}{l} \leftarrow \text{quotient} \\ \\ \\ \leftarrow \text{remainder} \end{array}$$

Thus $156 = 18 \cdot 8 + 12$ and hence
$\gcd(156, 18) = \gcd(18, 12)$ by Lemma 4.8.2.

Example 6 – *Solution*

3. Divide 18 by 12:

$$\begin{array}{r} 1 \\ 12\overline{)\,18} \\ \underline{12} \\ 6 \end{array} \quad \begin{array}{l} \leftarrow \text{quotient} \\ \\ \\ \leftarrow \text{remainder} \end{array}$$

Thus $18 = 12 \cdot 1 + 6$ and hence $\gcd(18, 12) = \gcd(12, 6)$ by Lemma 4.8.2.

Example 6 – *Solution*

4. Divide 12 by 6:

$$\begin{array}{r} 2 \leftarrow \text{quotient} \\ 6\overline{)\,12} \\ \underline{12} \\ 0 \leftarrow \text{remainder} \end{array}$$

Thus $12 = 6 \bullet 2 + 0$ and hence $\gcd(12, 6) = \gcd(6, 0)$ by Lemma 4.8.2.

Example 6 – *Solution*

cont'd

Putting all the equations above together gives

$$\begin{aligned}
\gcd(330, 156) &= \gcd(156, 18) \\
&= \gcd(18, 12) \\
&= \gcd(12, 6) \\
&= \gcd(6, 0) \\
&= 6 \qquad \text{by Lemma 4.8.1.}
\end{aligned}$$

Therefore, gcd(330, 156) = 6.

The Euclidean Algorithm

The following is a version of the Euclidean algorithm written using formal algorithm notation.

Algorithm 4.8.2 Euclidean Algorithm :
[Given two integers A and B with A > B ≥ 0, this algorithm computes gcd(A, B). It is based on two facts:

1. $\gcd(a, b) = \gcd(b, r)$ *if a, b, q, and r are integers with*

$$a = b \cdot q + r \text{ and } 0 \leq r < b.$$

2. $\gcd(a, 0) = a$.*]*

The Euclidean Algorithm

Input: *A*, *B [integers with* $A > B \geq 0$*]*

Algorithm Body:

$$a := A, b := B, r := B$$

[If $b \neq 0$*, compute a mod b, the remainder of the integer division of a by b, and set r equal to this value. Then repeat the process using b in place of a and r in place of b.]*

while $(b \neq 0)$

$$r := a \bmod b$$

The Euclidean Algorithm

[The value of a mod b can be obtained by calling the division algorithm.]

$a := b$

$b := r$

end while

[After execution of the **while** *loop,* $\gcd(A, B) = a$*.]*

$\gcd := a$

Output: gcd *[a positive integer]*