Software Development (CS2500)

Lecture 16: The Euclidean Algorithm

M.R.C. van Dongen

November 8, 2010

Contents

1 The Euclidean Algorithms

This section studies the oldest known algorithm: the Euclidean Algorithm—more than 2000 years old. We shall start with some basic definitions. Next we shall study a naive algorithm for computing greatest common divisors. We continue with the *Euclidean Algorithm*, which is much more efficient. Next, we shall study the *Extended Euclidean Algorithm* and an interesting application: computing multiplicative inverses modulo prime numbers. We shall conclude by studying an interesting fact about prime numbers.

1.1 Basic Definitions

Let s be a non-negative integer. An integer, $a \geq 0$, is *a multiple of* s if $a = s \times t$, for some integer t. We write $s \mid a$ if a is a multiple of s. We write $s \nmid a$ is a is not a multiple of s. By "analogy" we say that s *divides* a if $s \mid a$. The following are some examples:

- $1 \mid a$ for every $a \in \mathbb{N}$. Proof: $s = 1$ take $t = a$.

- $2 \mid a$ for every even $a \in \mathbb{N}$. Proof: $s = 2$ take $t = a/2$.
- $2 \nmid a$ for every odd $a \in \mathbb{N}$. Proof: $s = 2$ but $t = a/2$ isn't an integer.
- $s \mid 0$ for every $s \in \mathbb{N}$. Proof: take $t = 0$.
- Let a and s be positive integers. If $a \,\%\, s \neq 0$, then $s \nmid a$. Proof: $t = a/s$ is not an integer.

Definition 1 (Greatest Common Divisor)**.** Let a and b be non-negative integers. The *greatest common divisor* of a and b is the largest integer g such that $g \mid a$ and $g \mid b$. The greatest common divisor of a and b is denoted $\gcd(a, b)$.

Note: $\gcd(i, 0) = i$ for all non-negative integers i. Proof: clearly, $i \mid 0$ and $i \mid i$. Next note that $\gcd(i, 0)$ cannot exceed i, so i is the largest possible common divisor.

1.2 Naive Gcd Algorithms

This section studies an obvious algorithm for computing greatest common divisors. The algorithm is based on the factorisation-based algorithm for bringing fractions to lowest terms. The following is the a possible implementation of the algorithm. However, the algorithm is hopelessly inefficient.

Don't Try this at Home

```
public static int factorisationBasedGcd( int a, int b ) {
    int gcd;

    if (a == 0) {
        gcd = b;
    } else if (b == 0) {
        gcd = a;
    } else {
        gcd = 1;
        int factor = 2;
        while ((a >= factor) && (b >= factor)) {
            if (((a % factor) == 0) && ((b % factor) == 0)) {
                a /= factor;
                b /= factor;
                gcd *= factor;
            } else {
               factor ++;
            }
        }
    }

    return gcd;
}
```

The termination and correctness proofs are left as an exercise for the reader.

1.3 The Euclidean Algorithm

The *Euclidean Algorithm* is one of the oldest known algorithms. Given two non-negative integers a and b it computes the GCD of a and b. The algorithm is so important that you must know it for the exam. Proving termination is trivial, so you also have to know that. The only thing which you don't need to know is a correctness proof. The following is an implementation of the algorithm in Java.

```java
/**
 * Compute greatest common divisor of two nonnegative integers.
 *
 * @param a a nonnegative integer.
 * @param b a nonnegative integer.
 * @return the greatest common divisor of {@code a} and {@code b}.
 */
public static int gcd( int a, int b ) {
    while (b != 0) {
        final int r = a % b;
        a = b;
        b = r;
    }
    return a;
}
```

Theorem 2 (The Euclidean Algorithm Terminates)**.** *It is not difficult to see that the algorithm terminates. For example, assume it doesn't. Let b_i be the value of* b *in the ith iteration. Then this gives rise to the infinite sequence*

$$b_1 > b_2 > b_3 > \dots .$$

No such sequence exists because each b_i is a positive integer.

The correctness proof is not much more complicated, but you don't need to know it for the exam. For simplicity, we use the following recursive version of the algorithm. The variable q isn't needed but it helps simplify the proof. Make sure you understand why this algorithm is equivalent.

```java
public static int gcd( int a, int b ) {
    final int gcd;

    if (b != 0) {
        final int q = a / b;
        final int r = a % b; // a == r + q * b AND r == a - q * b.
        gcd = gcd( b, r );
    } else {
        gcd = a;
    }

    return gcd;
}
```

Correctness of Euclidean Algorithm. *Assume* $b = 0$. Clearly the algorithm computes $\gcd(a, b)$. *Assume* $b \neq 0$. Let $g = \gcd(a, b)$ and let $G = \gcd(b, r)$. First notice that $r = a - q \times b$. Since $g \mid a$ and $g \mid b$, it follows that $g \mid r$ and that $g \leq G$. Next notice that $a = r + q \times b$. Since $G \mid b$ and $G \mid r$, it follows that $G \mid a$ and that $g \geq G$. We may complete the proof by observing that $g \leq G$ and $g \geq G$. Therefore $g = G$. □

1.4 Simplifying Fractions

In this section we shall study an application of the Euclidean Algorithm: simplifying factions.

Let $a \geq 0$ and $b > 0$ be two integers. The algorithm for reducing the fraction $\frac{a}{b}$ to lowest terms may be done with the aid of the Euclidean Algorithm. To simplify the fraction, you compute

$$\frac{a/\gcd(a, b)}{b/\gcd(a, b)}.$$

1.5 Interlude: Finite Fields

A *field* is a structure where addition, subtraction, division, and multiplication work "as expected". For example, the rationals, reals, and complex numbers are fields. $\mathbb{N}$ is not a field because is $x \in \mathbb{N}$ and $y \in \mathbb{N}$ then $x - y$ may not be in $\mathbb{N}$. In technical terms we say that $\mathbb{N}$ is not closed under subtraction. Likewise $\mathbb{Z}$ is not a field because it is not closed under division. A field is *finite* if it has finitely many members.

An example of a finite field is arithmetic modulo a prime number p. Addition, subtraction, and multiplication work as usual, but we make sure that the result is in the range $\{0, \ldots, p - 1\}$ by adding a suitable multiple of p. For division we need a multiplicative inverse x^{-1} for each $x \neq 0$. This inverse should satisfy $x \times x^{-1} \equiv 1 \pmod{p}$. It is guaranteed that such an x^{-1} exists. Next we define $y/x = y \times x^{-1}$.

Tables 1–4 list the Cayley tables for addition, additive inverse, multiplication, and multiplicative inverse of a finite field: arithmetic modulo 3. Strange as it may seem, Table 4 shows that $2^{-1} = 2$. This follows from the fact that $x \times x^{-1} \equiv 1 \pmod{p}$ for all $x \neq 0$ and for $x = 2$ in particular.

$+$	0	1	2
0	0	1	2
1	1	2	0
2	2	0	1

Table 1: Cayley table for addition modulo 3.

x	$-x$
0	0
1	2
2	1

Table 2: Cayley table for additive inverse modulo 3.

$\times$	0	1	2
0	0	0	0
1	0	1	2
2	0	2	1

Table 3: Cayley table for multiplication modulo 3.

1.6 The Extended Euclidean Algorithm

It is well known that if a and b are non-negative integers, then

$$\gcd(a, b) = s \times a + t \times b\,,$$

for suitably chosen integers s and t. The *Extended Euclidean Algorithm* computes $\gcd(a, b)$ as well as the integers s and t. Effectively, the algorithm is the same as the Euclidean Algorithm. However, this time it expresses a_i and b_i as linear combinations of a_0 and b_0. Here we use the notation $\langle \texttt{var} \rangle_i$ for the value of $\langle \texttt{var} \rangle$ in the ith iteration and $\langle \texttt{var} \rangle_0$ for the initial value of $\langle \texttt{var} \rangle$. Initially, we have:

$$\begin{bmatrix} a_1 \\ b_1 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} a_0 \\ b_0 \end{bmatrix}.$$

The notation $\begin{bmatrix} a_i \\ b_i \end{bmatrix} = \begin{bmatrix} s_i & t_i \\ u_i & v_i \end{bmatrix} \begin{bmatrix} a_0 \\ b_0 \end{bmatrix}$ is shorthand notation for the following two equalities: $a_i = s_i \times a_0 + t_i \times b_0$ and $b_i = u_i \times a_0 + v_i \times b_0$. In the ith iteration of the extended Algorithm we have:

$$\begin{bmatrix} a_i \\ b_i \end{bmatrix} = \begin{bmatrix} s_i & t_i \\ u_i & v_i \end{bmatrix} \begin{bmatrix} a_0 \\ b_0 \end{bmatrix}.$$

x	x^{-1}
1	1
2	2

Table 4: Cayley table for multiplicative inverse modulo 3.

We also have $a_{i+1} = b_i$ and $b_{i+1} = a_i - a_i \times q_i$. This translates to:

$$\begin{bmatrix} a_{i+1} \\ b_{i+1} \end{bmatrix} = \begin{bmatrix} u_i & v_i \\ s_i - q_i \times u_i & t_i - q_i \times v_i \end{bmatrix} \begin{bmatrix} a_0 \\ b_0 \end{bmatrix}.$$

This algorithm is not examinable.

Table 5 provides an example.

a_i	b_i	q_i	s_i	t_i	u_i	v_i
3	5	0	1	0	0	1
5	3	1	0	1	1	0
3	2	1	1	0	−1	1
2	1	2	−1	1	2	−1
1	0		2	−1	−5	3

Table 5: Simulating the Extended Euclidean Algorithm.

The following is the core of the algorithm. For sake of presentation, some lines have two statements. Note that the first two statements are not needed, but they allow us to write the invariants. Note that the first four statements in the `while` statement are the same as in the Euclidean Algorithm. The remaining statements are for the additional computations.

```java
A = a;
B = b;
s = 1; t = 0; // a = s * A + t * B
u = 0; v = 1; // b = u * A + v * B
while (b != 0) {
    q = a / b;
    r = a % b;
    a = b;
    b = r;
    s1 = s;
    t1 = t;
    s = u;
    t = v;          // a = s * A + t * B
    u = s1 - q * u;
    v = t1 - q * v; // b = u * A + v * B
}
```

Before studying an application of the Extended Euclidean Algorithm, it is recalled that an integer, $p > 1$, is a *prime* if $i \nmid p$, for all integers i such that $1 < i < p$. It can be shown that if p is a prime and $1 \leq i < p$ an integer, then $\gcd(i, p) = 1$. Note that if $p \,\%\, i \neq 0$ then $i \nmid p$.

A common application of the Extended Euclidean Algorithm is to compute multiplicative inverses over finite fields. Let p be a prime number. In "arithmetic" modulo p, we represent numbers as non-negative integers less than p. Implementing modular addition, subtraction, and multiplication are straightforward. We can implement division with the Extended Euclidean Algorithm.

For example, let $0 \leq x < p$ be an integer. Dividing by x is equivalent to multiplying by x^{-1}. Since p is a prime, we know that $\gcd(x, p) = 1$. Therefore, applying the algorithm finds s and t such that

$$s \times x + t \times p = 1\,.$$

It immediately follows that $s \times x \equiv 1 \pmod{p}$. Therefore, $s \equiv x^{-1} \pmod{p}$.

For example, let $p = 5$ and let $x = 3$. We want to find an integer y such that $x \times y \equiv 1 \pmod{p}$. (We require that $0 \leq y < p$.) Applying the algorithm gives us $s = 2$ and $t = -1$, so $y = 2$. (Note that it is not always guaranteed that $0 \leq s < p$, so shifting to $\{0, \ldots, p-1\}$ may be required.) Lo and behold: $x \times y \equiv 6 \pmod{5} \equiv 1 \pmod{5}$.

1.7 Interesting Facts

In this section we shall look at some interesting facts about prime numbers.

Proposition 1. *Let p be a prime and let $0 < i < p$ and $0 \leq k \leq p$ be integers. Then $(i \times k) \,\%\, p = 0$ if and only if $k = 0$ or $k = p$.*

Proof. It is recalled that $(i \times k) \,\%\, p = 0$ if and only if $i \times k \,||\, p$. Since p is a prime, $i \times k \mid p$ if and only if $i \mid p$ or $k \mid p$. We have $i \nmid p$ because $0 < i < p$ and because p is a prime. Likewise, $k \nmid p$ if $0 < k < p$. However, if $k \in \{0, p\}$, then $(i \times k) \,\%\, p = 0$. □

In the following two lectures we shall study some algorithms that exploit Proposition 1. The following are the preliminaries.

Let p be a prime. Let's assume we have p squares: s_0, s_1, ..., s_{p-1}. Let's pick an initial square: s_i. We know that we can visit all squares using the sequence

$$s_i\,, s_{(i+1)\,\%\,p}\,, s_{(i+2)\,\%\,p}\,, \cdots\,, s_{(i+p-1)\,\%\,p}\,.$$

The theorem tells us we can also visit all squares using other offsets than 1. For example, if $p = 5$ and $i = 0$, then each of the following work:

$$[\,s_0\,, s_1\,, s_2\,, s_3\,, s_4\,]\,, \qquad [\,s_0\,, s_2\,, s_4\,, s_1\,, s_3\,]\,, \qquad [\,s_0\,, s_3\,, s_1\,, s_4\,, s_2\,]\,,$$

and

$$[\,s_0\,, s_4\,, s_3\,, s_2\,, s_1\,]\,.$$

It should also work for other initial positions i.

2 Modular Arithmetic

In this section we shall implement a class for arithmetic modulo a prime number.

The following is a rough design of the class.

```java
/**
 * Support for computations in finite fields.
 *
 * <PAR> The number of elements in the field should be a prime. </PAR>
 *
 * @author M. R. C. van Dongen
 */
public class FiniteField {
    /**
     * The size of the field.
     */
    private final int prime;

    /**
     * Basic constructor.
     *
     * @param size the size of the field.
     */
    public FiniteField( int size ) {
        prime = size;
    }
    ⋮
    @Override
    public String toString( ) {
        return "FiniteField[ prime = " + prime + " ]";
    }
}
```

The following method "scales" and "shifts" a given `int` to the range $\{0, \ldots, p-1\}$, where p is the size of the field. In technical terms, we want a representative of the `int` which is equivalent to the `int` modulo p. Here, two numbers a and b are equivalent modulo p if $a - b$ is a multiple of p. A non-negative number is a representative if it is less than p. Because of Java's (odd) rules for division, we need to be careful when `number` is negative: returning `number % prime` doesn't work!

```Java
/**
 * Compute representative of given integer.
 *
 * @param number any integer.
 * @return the unique integer, {@code rep}, such that
 *         {@code 0 <= rep} and {@code rep < prime} and
 *         {@code number == rep} modulo {@code prime}.
 */
public int representative( int number ) {
    return (prime + number % prime) % prime;
}
```

The following method add two `int`s and returns the result modulo `prime`.

```Java
/**
 * Add two integers modulo size of the field.
 *
 * @param a the first operand.
 * @param b the second operand.
 * @return the sum of {@code a} and {@code b} modulo {@code prime}.
 */
public int add( int a, int b ) {
    return representative( representative( a ) + representative( b ) );
}
```

We can implement subtraction in terms of addition and "negation", where negating a number, n, means computing a number, m, such that $n + m \equiv 0 \pmod{p}$.

```Java
/**
 * Subtract two integers modulo size of the field.
 *
 * @param a the first operand.
 * @param b the second operand.
 * @return the difference of {@code a} and {@code b} modulo {@code prime}.
 */
public int subtract( int a, int b ) {
    return add( a, additiveInverse( b ) );
}

/**
 * Compute the additive inverse of a given number modulo the size
 * of the field.
 *
 * @param a the integer.
 * @return {@code -a} modulo {@code prime}.
 */
public int additiveInverse( int a ) {
    return representative( - a );
}
```

Multiplication is straightforward.

```Java
/**
 * Multiply two integers modulo size of the field.
 *
 * @param a the first operand.
 * @param b the second operand.
 * @return the product of @code a and @code b modulo @code prime.
 */
public int multiply( int a, int b ) {
    // Note that representative( a * b ) is more fragile because of overflow.
    return representative( representative( a ) * representative( b ) );
}
```

Division may be implemented using multiplication. The method `reciprocal( )` is for computing multiplicative inverses. This is where we need the Extended Euclidean algorithm.

```Java
/**
 * Divide two integers modulo size of the field.
 *
 * @param a the first operand.
 * @param b the second operand.
 * @return the quotient of {@code a} and {@code b} modulo {@code prime}.
 */
public int divide( int a, int b ) {
    return multiply( a, reciprocal( b ) );
}
```

The implementation of the method method `reciprocal( )` is for completes the implementation of the class. Some lines of the method are combined to keep the listing short. Strictly speaking this violates the coding conventions, but it improves the presentation.

```Java
/**
 * Compute the multiplicative inverse of number modulo the size
 * of the field.
 *
 * @param a the integer.
 * @return The integer {@code i} such that
 *         {@code 0 < i},
 *         {@code i < prime}, and
 *         {@code i * a == 1} modulo {@code prime}.
 */
public int reciprocal( int a ) {
    int b, q, r, s, t, u, v, s1, t1;
    s = 1; t = 0;
    u = 0; v = 1;
    b = prime;
    while (b != 0) {
        q = a / b; r = a % b;
        a = b; b = r;
        s1 = s; t1 = t;
        s = u; t = v;
        u = s1 - q * u; v = t1 - q * v;
    }
    return representative( s );
}
```

3 For Wednesday

Study the notes.