Lecture 5: Finite Fields (PART 2)

PART 2: Modular Arithmetic

Theoretical Underpinnings of Modern Cryptography

Lecture Notes on "Computer and Network Security"

by Avi Kak (kak@purdue.edu)

January 18, 2018
8:29am

Goals:

- To review modular arithmetic
- To present Euclid's GCD algorithms
- To present the prime finite field Z_p
- To show how Euclid's GCD algorithm can be extended to find multiplicative inverses
- **Perl and Python implementations for calculating GCD and multiplicative inverses**

CONTENTS

5.1: MODULAR ARITHMETIC NOTATION

- Given **any** integer a and a **positive** integer n, and given a division of a by n that leaves the remainder between 0 and $n-1$, both inclusive, we define

$$a \; mod \; n$$

to be **the remainder**. Note that the remainder **must** be between 0 and $n-1$, both ends inclusive, even if that means that we must use a negative quotient when dividing a by n.

- We will call two integers a and b to be **congruent modulo** n if

$$a \; mod \; n \quad = \quad b \; mod \; n$$

- Symbolically, we will express such a **congruence** by

$$a \quad \equiv \quad b \pmod{n}$$

- Informally, a congruence may also be displayed as:

$$a \;=\; b \;\pmod{n}$$

and even

$$a \;=\; b \mod n$$

as long as it is understood that we are talking about a and b being equal only in the sense that their remainders obtained by subjecting them to modulo n division are exactly the same.

- We say a non-zero integer a is a **divisor** of another integer b provided the remainder is zero when we divide b by a. That is, when $b = ma$ for some integer m.

- When a is a divisor of b, we express this fact by $a \mid b$.

5.1.1: Examples of Congruences

- Here are some congruences modulo 3:

$$\begin{aligned} 7 &\equiv 1 \pmod 3 \\ -8 &\equiv 1 \pmod 3 \\ -2 &\equiv 1 \pmod 3 \\ 7 &\equiv -8 \pmod 3 \\ -2 &\equiv 7 \pmod 3 \end{aligned}$$

- Here is one way of seeing the above congruences (for mod 3 arithmetic):

```
...  0  1  2  0  1  2  0  1  2  0  1  2  0  1  2  0  1  2  0  1  2  0  ...
...- 9 -8 -7 -6 -5 -4 -3 -2 -1  0  1  2  3  4  5  6  7  8  9  10 11 12 ...
```

  where each entry in the upper line is the output of **modulo 3** arithmetic for the corresponding entry in the lower line. Note that the lower line depicts *all* integers – positive, negative, and zero – in the form of a sequence of increasingly larger numbers. [Pause for a moment and reflect on the fact that whereas $(7 \ mod \ 3) = 1$ on the positive side of the integers, on the negative side we have $(-7 \ mod \ 3) = 2$.]

- As you can see, the **modulo n** arithmetic maps all integers into the set $\{0, 1, 2, 3, ..., n-1\}$.

5.2: MODULAR ARITHMETIC OPERATIONS

- As mentioned on the previous page, **modulo n** arithmetic maps all integers into the set $\{0, 1, 2, 3, ..., n-1\}$.

- With regard to the modulo n arithmetic operations, the following equalities are easily shown to be true:

$$\begin{aligned}
[(a \bmod n) + (b \bmod n)] \bmod n &= (a + b) \bmod n \\
[(a \bmod n) - (b \bmod n)] \bmod n &= (a - b) \bmod n \\
[(a \bmod n) \times (b \bmod n)] \bmod n &= (a \times b) \bmod n
\end{aligned}$$

  with ordinary meanings ascribed to the arithmetic operators.

- To prove any of the above equalities, you write a as $mn + r_a$ and b as $pn + r_b$, where r_a and r_b are the **residues** (the same thing as **remainders**) for a and b, respectively. You substitute for a and b on the right hand side and show you can now derive the left hand side. Note that r_a is $a \bmod n$ and r_b is $b \bmod n$.

- For **arithmetic modulo n**, let Z_n denote the set

$$Z_n \quad = \quad \{0,\ 1,\ 2,\ 3,\ .....,\ n-1\}$$

Z_n is **the set of remainders** in arithmetic modulo n. It is officially called the **set of residues**.

- Finally, here is a useful memaid (short for "memory aid"): In *mod n* arithmetic, any time you see n or any of its multiples, think 0. That is, the numbers n, $2n$, $3n$, $-n$, $-2n$, etc., are exactly the same number as 0.

- Here is another memaid that you are going to need when we talk about public-key crypto in Lecture 12: Anytime you see the number -1 in *mod n* arithmetic, you should think $n-1$. That is, the number $n-1$ is exactly the same thing as the number -1 in *mod n* arithmetic.

- A personal note: I consider memaids as convenient mechanisms for what psychologists refer to as *memory offloading*. Normally, as you encounter an engineering or a math detail, in order for you to accept that detail as credible, your brain needs to bring up all the supporting arguments justifying the detail. While initially this happens consciously, ultimately it becomes a subconscious process. Regardless of whether you do it consciously or subcon-

sciously, you can speed up the process by identifying certain facts as *memaids* and letting your brain use those as jumping off points for more elaborate justifications.

5.3: THE SET Z_n AND ITS PROPERTIES

- Reall the definition of Z_n as the set of remainders in modulo n arithmetic.

- Let's now consider the set Z_n along with the following two binary operators defined for the set: (1) modulo n addition; and (2) modulo n multiplication. The elements of Z_n obey the following properties vis-a-vis these operators:

Commutativity:

$$\begin{aligned} (w \;+\; x)\; mod\; n &= (x \;+\; w)\; mod\; n \\ (w \;\times\; x)\; mod\; n &= (x \;\times\; w)\; mod\; n \end{aligned}$$

Associativity:

$$\begin{aligned} [(w \;+\; x) \;+\; y]\; mod\; n &= [w \;+\; (x \;+\; y)]\; mod\; n \\ [(w \;\times\; x) \;\times\; y]\; mod\; n &= [w \;\times\; (x \;\times\; y)]\; mod\; n \end{aligned}$$

Distributivity of Multiplication over Addition:

$$[w \;\times\; (\; x \;+\; y)]\; mod\; n = [(w \;\times\; x) + \;(w \;\times\; y)]\; mod\; n$$

Existence of Identity Elements:

$$\begin{aligned} (0 \ + \ w) \ mod \ n \quad &= \quad (w \ + \ 0) \ mod \ n \\ (1 \ \times \ w) \ mod \ n \quad &= \quad (w \ \times \ 1) \ mod \ n \end{aligned}$$

Existence of Additive Inverses:

For each $w \in Z_n$, there exists a $z \in Z_n$ such that

$$w \ + \ z \quad = \quad 0 \ mod \ n$$

5.3.1: So What is Z_n?

- Is Z_n a group? If so, what is the group operator? [The group operator is the modulo n addition.]

- Is Z_n an abelian group?

- Is Z_n a ring?

- Actually, Z_n is a commutative ring. Why? [See the previous lecture for why.]

- You could say that Z_n is more than a commutative ring, but not quite an integral domain. What do I mean by that? [Because Z_n contains a multiplicative identity element. Commutative rings are not required to possess multiplicative identities.]

- Why is Z_n not an integral domain? [Even though Z_n possesses a multiplicative identity, it does NOT satisfy the other condition of integral domains which says that if $a \times b = 0$ then either a or b must be zero. Consider modulo 8 arithmetic. We have $2 \times 4 = 0$, which is a clear violation of the second rule for integral domains.]

- Why is Z_n not a field?

5.3.2: Asymmetries Between Modulo Addition and Modulo Multiplication Over Z_n

- For every element of Z_n, there exists an **additive inverse** in Z_n. But there does not exist a **multiplicative inverse** for every non-zero element of Z_n.

- Shown below are the additive and the multiplicative inverses for **modulo 8** arithmetic:

Z_8	:	0	1	2	3	4	5	6	7
additive inverse	:	0	7	6	5	4	3	2	1
multiplicative inverse	:	-	1	-	3	-	5	-	7

- Note that the **multiplicative inverses** exist for only those elements of Z_n that are **relatively prime** to n. Two integers are relatively prime to each other if the integer 1 is their only common positive divisor. More formally, two integers a and b are relatively prime to each other if $gcd(a,\ b) = 1$ where gcd denotes the **Greatest Common Divisor**.

- The following property of **modulo n addition** is the same as for ordinary addition:

$$(a \;+\; b) \;\equiv\; (a \;+\; c)\; (mod\; n) \qquad implies \qquad b \;\equiv\; c\; (mod\; n)$$

  But a similar property is **NOT** obeyed by **modulo n multiplication**. That is

$$(a \;\times\; b) \;\equiv\; (a \;\times\; c)\; (mod\; n) \qquad does\; not\; imply \qquad b \;\equiv\; c\; (mod\; n)$$

  unless a and n are **relatively prime** to each other.

- That the **modulo n addition** property stated above should hold true for all elements of Z_n follows from the fact that the **additive inverse** $-a$ exists for every $a \in Z_n$. So we can add $-a$ to both sides of the equation to prove the result.

- To prove the same result for **modulo n multiplication**, we will need to multiply both sides of the second equation above by the multiplicative inverse a^{-1}. But, as you already know, not all elements of Z_n possess multiplicative inverses.

- Since the existence of the multiplicative inverse for an element a of Z_n is predicated on a being **relatively prime** to n and since the answer to the question whether two integers are relatively prime to each other depends on their **greatest common divisor** (GCD), let's explore next the world's most famous algorithm for finding the GCD of two integers.

- The gcd algorithm that we present in the next section is by Euclid who is considered to be the father of geometry. He was born around 325 BC.

5.4: EUCLID'S METHOD FOR FINDING THE GREATEST COMMON DIVISOR OF TWO INTEGERS

- We will now address the question of how to efficiently find the GCD of any two integers. [When there is a need to find the GCD of two integers in actual computer security algorithms, the two integers are always extremely large — much too large for human comprehension, as you will see in the lectures that follow.]

- Euclid's algorithm for GCD calculation is based on the following observations [Recall from Section 5.1 that the notation $b|a$ means that "b is a divisor of a." That is, when we divide a by b, we are left with zero remainder.]:

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

  - $if \quad b|a \quad then \quad gcd(\ a,\ b) \quad = \quad b$

  - $gcd(\ a,\ 0) \quad = \quad a \quad since\ it\ is\ always\ true\ that \quad a|0$

  - Assuming without loss of generality that a is larger than b, it can be shown that (See Section 5.4.3 for proof)

$$gcd(\ a,\ b) \quad = \quad gcd(\ b,\ a\ mod\ b\ )$$

The critical thing to note in the above recursion is that the right hand side of the equation is an easier problem to solve than the left hand side. While the largest number on the left is a, the largest number on the right is b, which is smaller than a.

- The above recursion is at the heart of Euclid's algorithm (now over 2000 years old) for finding the GCD of two integers. As already noted, the call to $gcd()$ on the right in Euclid's recursion is an easier problem to solve than the call to $gcd()$ on the left.

- As a fun aside, some people are visually bothered by the boundary condition $gcd(a, 0) = a$ on the recursion since, at first reflection, it appears to violate your expectation that $gcd(a, b)$ will not exceed the smaller of the two integers involved. (For example, you fully expect that $gcd(123541, 23)$ will not exceed the smaller number 23.) But then 0 is no ordinary integer.

5.4.1: Steps in a Recursive Invocation of Euclid's GCD Algorithm

- To elaborate on the recursive calculation of GCD in Euclid's algorithm:

$$
\begin{aligned}
& gcd(b_1, b_2) & & & \textit{assume } b_1 > b_2 \\
& \quad = gcd(b_2, b_1 \ mod \ b_2) & = gcd(b_2, b_3) & & \textit{simpler since } b_2 > b_3 \\
& \quad = gcd(b_3, b_2 \ mod \ b_3) & = gcd(b_3, b_4) & & \textit{simpler still} \\
& \quad = gcd(b_4, b_3 \ mod \ b_4) & = gcd(b_4, b_5) & & \textit{simpler still} \\
& \quad .... & .... & & \\
& \quad .... & .... & & \\
& \quad until \ \ b_{m-1} \ \ mod \ \ b_m \ == \ 0 \ \ then \ \ gcd(b1, b2) \ = \ b_m & & &
\end{aligned}
$$

- Although we assumed $b_1 > b_2$ in the recursion illustrated above, note that the algorithm works for any two non-negative integers $b1$ and $b2$ regardless of which is larger. If the first integer is smaller than the second integer, the first iteration will swap the two.

5.4.2: An Example of Euclid's GCD Algorithm in Action

```
gcd( 70, 38 )
    =   gcd( 38, 32 )
    =   gcd( 32, 6 )
    =   gcd( 6, 2 )
    =   gcd( 2, 0 )

Therefore,  gcd( 70, 38 )  = 2
```

Another Example (for relatively prime pair of integers):

```
gcd( 8, 17 ):
   =  gcd( 17, 8 )
   =  gcd( 8, 1 )
   =  gcd( 1, 0 )

Therefore,  gcd( 8, 17 )  = 1
```

When the smaller of the two arguments in a call to *gcd*() is 1 (which happens when the two starting numbers are relatively prime), there is no need to go to the last step in which the smaller of the two arguments is 0.

Here is an example of Euclid's GCD algoirthm for two moderately large numbers:

```
gcd( 40902, 24140 )
    =   gcd( 24140, 16762 )
    =   gcd( 16762, 7378 )
    =   gcd( 7378, 2006 )
    =   gcd( 2006, 1360 )
    =   gcd( 1360, 646 )
    =   gcd( 646, 68 )
    =   gcd( 68, 34 )
    =   gcd( 34, 0 )

Therefore,  gcd( 40902, 24140 )  = 34
```

5.4.3: Proof of Euclid's GCD Algorithm

The proof of Euclid's algorithm is based on the following observation:

- Given any two non-negative integers a and b, with $a > b$, we can write $a = qb + r$ for some non-negative quotient integer q and some non-negative remainder integer r.

- Every common divisor of a and b must therefore be a common divisor of $qb + r$ and b. Since the product qb is trivially divisible by b, it is surely the case that every common divisor of a and b is a common divisor of r and b.

- That is, all **common** divisors for a and b are the same as those for b and r.

- Since $gcd(a, b)$ is one of those common divisors, then it must be the case that $gcd(a, b) = gcd(b, r)$.

5.4.4: Implementing the GCD Algorithm in Perl and Python

- The Python implementation of Euclid's algorithm shown below couldn't be simpler. The cool thing about this script is that the two-line `while` loop takes care of all of the boundary conditions that terminate the recursion, these being $gcd(a, a) = a$, $gcd(a, 0) = gcd(0, a) = a$, and $gcd(a, b) = b$ if b divides a without leaving a non-zero remainder:

```
#!/usr/bin/env python

##  GCD.py

import sys
if len(sys.argv) != 3:
    sys.exit("\nUsage:   %s  <integer>  <integer>\n" % sys.argv[0])

a,b = int(sys.argv[1]),int(sys.argv[2])

while b:
    a,b = b, a%b

print("\nGCD: %d\n" % a)
```

- The calls shown on the left return the answer shown on the right:

```
GCD.py 15 18                    =>          GCD: 3

GCD.py 123456789 987654321      =>          GCD: 9
```

- And shown below is an equally simple Perl implementation. All the good things I said about the Python implementation apply just the same to the Perl implementation:

```
#!/usr/bin/env perl

##  GCD.pl
##  Avi Kak

use strict;
use warnings;

die "\nUsage:   $0  <integer>  <integer>\n" unless @ARGV == 2;

die "At least one of your numbers is too large! Use GCDWithBigInt.pl instead\n"
    if ($ARGV[0] > 0x7f_ff_ff_ff) or ($ARGV[1] > 0x7f_ff_ff_ff);

my ($a,$b) = @ARGV;
while ($b) {
    ($a,$b) = ($b, $a % $b);
}
print "\nGCD: $a\n\n";
```

This script behaves in exactly the same fashion as the Python script — as long as the integers involved are small enough to fit Perl's 4-byte representation for unsigned ints. That is the reason for the exception that is thrown in the second statement. For large integers, use the following script instead:

```
#!/usr/bin/perl -w

##  GCDWithBigInt.pl
##  Avi Kak

use strict;
use Math::BigInt;

die "\nUsage:   $0  <integer>  <integer>\n" unless @ARGV == 2;
```

```
my ($a,$b) = @ARGV;

$a = Math::BigInt->new("$a");
$b = Math::BigInt->new("$b");

while ($b->is_pos()) {
    ($a,$b) = ($b, $a->copy()->bmod($b));
}

print "\nGCD: $a\n\n";
```

- So if you call

```
GCDWithBigInt.pl  83975398475398749837499 2948576793949587674444
```

you will get the answer "GCD: 23". As you know, with Python, you do not have to do anything special for calculating with large numbers since it natively knows how to deal with numbers of arbitrary size.

- There is an alternative approach to calculating the GCD of two integers that in some cases may prove faster. This method, explained in the rest of this subsection, is referred to as the **Binary GCD algorithm**. It is also known as the **Stein's algorithm** after Josef Stein who first published it in 1967.

- Just as the boundary conditions and the recursion in Euclid's GCD algorithm are best for a computer with direct hardware support for divisions and multiplications, the same in the binary GCD algorithm are meant for a computer (which is likely to be

an embedded device) that prefers to implement multiplications and division by appropriately shifting the binary code word representations of the integers. [**As you know, shifting a binary code word to the left by one bit position means multiplication by 2. Similarly, shifting by one bit position to the right means division by 2. Before you do the latter, you would want to make sure that you are dealing with an even integer, that is, with an integer whose LSB (least significant bit) is not set.**]

- The previously stated boundary conditions $gcd(a, a) = a$, and $gcd(a, 0) = gcd(0, a) = a$ also applies to the binary GCD algorithm. However, for a recursive implementation of the algorithm, we must now consider the following five cases:

  1. If both the integers a and b are even, meaning if the LSB for both integers is not set, then 2 is a common factor of the two integers. So $gcd(a, b) = 2 \times gcd(a/2, b/2)$. The new arguments $a/2$ and $b/2$ are obtained by shifting the binary word representations for each integer to the right by one bit position. The multiplication by 2 in the recursion is achieved by shifting to the left the *gcd* result returned by the recursive call.

  2. If a is even and b is odd, then $gcd(a, b) = gcd(a/2, b)$. So we shift a to the right by one bit position and call *gcd* again.

  3. If a is odd and b is even, then $gcd(a, b) = gcd(a, b/2)$. So we shift b to the right by one bit position and call *gcd* again.

4. If both a and b are odd and, at the same time, $a > b$, then we can show that the *gcd* recursion takes the following form $gcd(a, b) = gcd(a - b, b) = gcd((a - b)/2, b)$, where the first step is basically a rewrite of Euclid's original recursion and the second step a consequence of the fact that when both a and b are odd, their difference is even. As we mentioned above, when *gcd* is called with the first argument even and the second argument odd, we make a recursive call in which we divide the first argument by 2 and leave the second unchanged.

5. If both a and b are odd and, at the same time, $a < b$, then, reasoning in the same manner as in the previous step, we can show that the *gcd* recursion takes the following form $gcd(a, b) = gcd(b - a, a) = gcd((b - a)/2, a)$.

- Shown below is a Python implementation of the binary GCD algorithm:

```
#!/usr/bin/env python

##  BGCD.py

import sys
if len(sys.argv) != 3:
    sys.exit("\nUsage:   %s  <integer>  <integer>\n" % sys.argv[0])

a,b = int(sys.argv[1]),int(sys.argv[2])

def bgcd(a,b):
    if a == b: return a                                          #(A)
    if a == 0: return b                                          #(B)
    if b == 0: return a                                          #(C)
    if (~a & 1):                                                 #(D)
        if (b &1):                                               #(E)
            return bgcd(a >> 1, b)                               #(F)
        else:                                                    #(G)
            return bgcd(a >> 1, b >> 1) << 1                     #(H)
    if (~b & 1):                                                 #(I)
        return bgcd(a, b >> 1)                                   #(J)
    if (a > b):                                                  #(K)
```

```
        return bgcd( (a-b) >> 1, b)                                  #(L)
    return bgcd( (b-a) >> 1, a )                                     #(M)

gcdval = bgcd(a, b)
print("\nBGCD: %d\n" % gcdval)
```

The implementation shown uses Python's bitwise operators for the integer types. [The unary operator '~' inverts the bits in its argument integer, the binary operator '&' carries out a bitwise `and` of the two arguments, the operator '<<' does a non-circular left shift of the left-argument integer by the number of positions that correspond to the right argument, and, finally, the operator '>>' does the same for the right shifts.] The test in line (D) checks whether a is even and that in line (E) checks whether b is odd. The recursion in line (H) will only be invoked when both a and b are even. Note how we multiply the answer returned by the recursive call by 2 by shifting it to the left by one position.

- As to how the five enumerated steps shown prior to the implementation on the previous page map to the various code lines, the recursion called by Step 1 is in line (H), by Step 2 in line F, by Step 3 in line (J), by Step 4 in line (L), and, finally, by Step 5 in line (M).

- Try making calls like

```
BGCD.py 321451443876 1255547372888

GCD.py 321451443876 1255547372888
```

to make sure that the two different implementation for calculating the GCD return the same answer.

- Shown next is a Perl implementation for the BGCD algorithm. Its logic mirrors that of the Python script shown above.

```
#!/usr/bin/perl -w

##  BGCD.pl

use strict;

die "\nUsage:   $0  <integer>  <integer>\n" unless @ARGV == 2;

my ($a,$b) = @ARGV;

my $gcdval = bgcd($a,$b );
print "\nBGCD: $gcdval\n\n";

sub bgcd {
    my ($a,$b) = @_;
    return $a if $a == $b;                                      #(A)
    return $b if $a == 0;                                       #(B)
    return $a if $b == 0;                                       #(C)
    if (~$a & 1) {                                              #(D)
        if ($b & 1) {                                           #(E)
            return bgcd($a >> 1, $b);                           #(F)
        } else {                                                #(G)
            return bgcd($a >> 1, $b >> 1) << 1;                 #(H)
        }
    }
    return bgcd($a,$ b >> 1) if (~$b & 1);                      #(I)
    return bgcd( ($a - $b) >> 1, $b) if ($a > $b);              #(J)
    return bgcd( ($b - $a) >> 1, $a );                          #(K)
}
```

5.5: PRIME FINITE FIELDS

- Earlier we showed that the set of remainders, Z_n is, in general, a commutative ring.

- The main reason for why, in general, Z_n is only a commutative ring and **not** a finite field is because not every element in Z_n is guaranteed to have a multiplicative inverse.

- In particular, as shown before, an element a of Z_n does **not** have a multiplicative inverse if a is not **relatively prime** to the modulus n.

- What if we choose the modulus n to be a **prime number**? (A prime number has only two divisors, one and itself.)

- For prime n, every non-zero element $a \in Z_n$ will be **relatively prime** to n. That implies that there will exist a **multiplicative inverse** for every non-zero $a \in Z_n$ for prime n.

- Therefore, Z_p is a **finite field** if we assume p denotes a **prime number**. Z_p is sometimes referred to as a **prime finite field**. Such a field is also denoted $GF(p)$, where GF stands for "Galois Field".

- Proving that, for prime p, every non-zero element of Z_p possess a unique MI (multiplicative inverse) is pretty straightforward. In a proof by contradiction, assume that a non-zero element $a \in Z_p$ possesses two *different* MIs b and c. That would imply $a \times b = 1 \; (mod \; p)$ and $a \times c = 1 \; (mod \; p)$. That would mean that $a \times (b - c) \equiv 0 \; (mod \; p) \equiv p \; (mod \; p)$. But that is impossible since the prime number p cannot be so factorized. The integer p only possesses only trivial factors, 1 and itself.

5.5.1: What Happened to the Main Reason for Why Z_n Could Not be an Integral Domain?

- Earlier, when we were looking at how to characterize Z_n, we said that, although it possessed a *multiplicative identity element*, it could not be an **integral domain** because Z_n allowed for the equality $a \times b = 0$ even for non-zero a and b. (Recall, 0 means the *additive identity element*.)

- If we have now decided that Z_p is a finite field for prime p because every element in Z_p has a unique multiplicative inverse, are we sure that we can now also guarantee that if $a \times b = 0$ then either a or b must be 0?

- Yes, we have that guarantee because $a \times b = 0$ for general Z_n occurs only when non-zero a and b are factors of the modulus n. When n is a prime, its only factors are 1 and n. So with the elements of Z_n being in the range 0 through $n-1$, the only time we will see $a \times b = 0$ is when either a is 0 or b is 0.

5.6: FINDING MULTIPLICATIVE INVERSES FOR THE ELEMENTS OF Z_p

- In general, to find the multiplicative inverse of $a \in Z_n$, we need to find an element $b \in Z_n$ such that

$$a \times b \equiv 1 \, (mod \, n)$$

- Based on the discussion so far, we can say that the multiplicative inverses exist for all $a \in Z_n$ for which we have

$$gcd(a, \, n) = 1$$

  When n equals a prime p, this condition will always be satisfied by all non-zero elements of Z_p.

- With regard to finding the value of the multiplicative inverse of a given integer a in modulo n arithmetic, we can do so with the help of Bezout's Identity that is presented below. The next section presents a proof of this identity. Subsequently, in Section 5.6.2, we will show how to actually use the identity for finding multiplicative inverses.

- In general, it can be shown that when a and n are **any** pair of positive integers, the following must always hold for some integers x and y (that may be positive or negative or zero):

$$gcd(a,\ n) \quad = \quad x \ \times \ a \ + \ y \ \times \ n \tag{1}$$

  This is known as the **Bezout's Identity**. For example, when $a = 16$ and $n = 6$, we have $gcd(16,\ 6) = 2$. We can certainly write: $2 = (-1) \times 16 + 3 \times 6 = 2 \times 16 + (-5) \times 6$. This shows that x and y do not have to be unique in Bezout's identity for given a and n.

5.6.1: Proof of Bezout's Identity

We will now prove that for a given pair of positive integers a and b, we have

$$gcd(a,\ b) \quad = \quad ax \quad + \quad by \tag{2}$$

for some positive or negative integers x and y.

- First define a set S as follows

$$S \quad = \quad \{am\ +\ bn\ \mid\ am\ +\ bn\ >\ 0,\ \ m,n \in N\} \tag{3}$$

  where N is the set of all integers. That is,

$$N \quad = \quad \{...., -3, -2, -1, 0, 1, 2, 3, ...\} \tag{4}$$

- Note that, by its definition, S can only contain *positive* integers. When $a = 8$ and $b = 6$, we have

$$S \quad = \quad \{2, 4, 6, 8....\} \tag{5}$$

  It is interesting to note that several pairs of (m, n) will usually result in the same element of S. For example, with $a = 8$ and

$b = 6$, the element 2 of S is given rise to by the following pairs of $(m, n) = (1, -1), (-2, 3), (4, -5), ...$

- Now let d denote the *smallest* element of S.

- Let's now express a in the following form

$$a = qd + r, \quad 0 \leq r < d \tag{6}$$

Obviously then,

$$\begin{aligned} r &= a \bmod d \\ &= a - qd \\ &= a - q(am + bn) \\ &= a(1 - qm) + b(-n) \end{aligned}$$

We have just expressed the residue r as a linear sum of a and b. But that is only possible if r equals 0. If r is not 0 but actually a non-zero integer *less than* d that it must be, that would violate the fact that d is the smallest positive linear sum of a and b.

- Since r is zero, it must be the case that $a = qd$ for some integer q. Similarly, we can prove that b is sd for some integer s. This proves that d is a common divisor of a and b.

- But how do we know that d is the GCD of a and b?

- Let's assume that some other integer c is also a divisor of a and b. Then it must be the case that c is a divisor of all linear combinations of the form $ma + nb$. Since d is of the form $ma + nb$, then c must be a divisor of d. This fact applies to any arbitrary common divisor c of a and b. That is, *every* common divisor c of a and b must also be a divisor of d.

- Hence it must be the case that d is the GCD of a and b.

5.6.2: Finding Multiplicative Inverses Using Bezout's Identity

- Given an a that is relatively prime to n, we must obviously have $gcd(a, n) = 1$. Such a and n must satisfy the following constraint for some x and y:

$$x \times a + y \times n = 1 \qquad (7)$$

  Let's now consider this equation **modulo n**. Since y is an integer, $y \times n \; mod \; n$ equals 0. Thus, it must be the case that, considered **modulo n**, x equals a^{-1}, the multiplicative inverse of a modulo n.

- Eq. (7) shown above gives us a strategy for finding the multiplicative inverse of an element a:

  – We use the same Euclid algorithm as before to find the $gcd(a, n)$,

  – but now at each step we write the expression in the form $a \times x + n \times y$ **for the remainder**

– eventually, before we get to the remainder becoming 0, when the remainder becomes 1 (which will happen only when a and n are relatively prime), x will automatically be the multiplicative inverse we are looking for.

- The next four subsections will explain the above algorithm in greater detail.

5.6.3: Revisiting Euclid's Algorithm for the Calculation of GCD

- Earlier in Section 5.4.1 we showed the following steps for a straightforward application of Euclid's algorithm for finding $gcd(b1,\ b2)$:

$$
\begin{aligned}
gcd(b_1, b_2) & \\
& = gcd(b_2, b_1 \ mod \ b_2) \ = \ gcd(b_2, b_3) \\
& = gcd(b_3, b_2 \ mod \ b_3) \ = \ gcd(b_3, b_4) \\
& = gcd(b_4, b_3 \ mod \ b_4) \ = \ gcd(b_4, b_5) \\
& .... \qquad\qquad .... \\
& .... \qquad\qquad .... \\
& until \ \ b_{m-1} \ \ mod \ \ b_m \ == \ 0 \ \ \ then \ \ gcd(b1, b2) \ = \ b_m
\end{aligned}
$$

- Next, let's make explicit the arithmetic operations required for carrying out the recursion at each step. This is shown on the next page.

- In the display shown below, what you see on the right of the vertical line makes explicit the arithmetic operations required for the computation of the remainders on the previous page:

$$
\begin{array}{lll|l}
gcd(b_1, b_2) & & & \textit{assume } b_1 > b_2 \\
= gcd(b_2, b_1 \ mod \ b_2) & = & gcd(b_2, b_3) & b_3 = b_1 - q_1 \times b_2 \\
= gcd(b_3, b_2 \ mod \ b_3) & = & gcd(b_3, b_4) & b_4 = b_2 - q_2 \times b_3 \\
= gcd(b_4, b_3 \ mod \ b_4) & = & gcd(b_4, b_5) & b_5 = b_3 - q_3 \times b_4 \\
.... & & .... & \\
.... & & .... & \\
gcd(b_{m-1}, b_m) & & & b_m = b_{m-2} - q_{m-2} \times b_{m-1}
\end{array}
$$

  until b_m is either 0 or 1.

- If $b_m = 0$ and b_{m-1} exceeds 1, then there does NOT exist a multiplicative inverse for b_1 in arithmetic modulo b_2. For example, $gcd(4, 2) = gcd(2, 0)$, therefore 4 has no multiplicative inverse modulo 2.

- If $b_m = 1$, then there exists a multiplicative inverse for b_1 in arithmetic modulo b_2. For examples, $gcd(3, 7) = gcd(7, 3) = gcd(3, 1)$ therefore there exists a multiplicative inverse for 3 modulo 7.

5.6.4: What Conclusions Can We Draw From the Remainders?

- The final remainder is always 0. By remainder we mean the second argument in the recursive call to $gcd()$ at each step.

- If the next to the last remainder is greater than 1, this remainder is the GCD of $b1$ and $b2$. Additionally, $b1$ and $b2$ are **NOT** relatively prime. **In this case, neither can have a multiplicative inverse modulo the other.**

- If the next to the last remainder is 1, the two input integers, $b1$ and $b2$, are relatively prime. In this case, $b1$ possesses a multiplicative inverse modulo $b2$.

5.6.5: Rewriting GCD Recursion in the Form of Derivations for the Remainders

- We will now focus solely on the remainders in the recusive steps shown on page 33.

- We will rewrite the calculation of the remainders shown to the right of the vertical line on page 33 in such a way that each remainder is a linear sum of the original integers $b1$ and $b2$.

- Note that before we get to the final remainder of 0, we are supposed to make sure that the remainder that comes just before the last is 1 (that is presumably the GCD of the two numbers if they are relatively prime):

```
gcd(b1, b2):

b3   =   b1  -  q1.b2

b4   =   b2  -  q2.b3
     =   b2  -  q2.(b1 - q1.b2)
     =   b2  -  q2.b1  +  q1.q2.b2
     =   -q2.b1  +  (1 + q1.q2).b2

b5   =   b3  -  q3.b4
     =   (b1 - q1.b2) - q3.( -q2.b1 + (1 + q1.q2).b2 )
```

```
          =   b1 + q2.q3.b1 - q1.b2 - q3.(1 + q1.q2).b2
          =   (1 + q2.q3).b1 - (q1 - q1.q2 - q3).b2
     .
     .

bm   =   (......).b1 ~~~ +  ~~~ (......). b2
```

- Stop when b_m is 1 (that will happen when b_1 and b_2 are co-prime). Otherwise, stop when b_m is 0, in which case there is no multiplicative inverse for b_1 modulo b_2.

- If you stopped because b_m is 1, then the multiplier of b_1 in the expansion for b_m is the multiplicative inverse of b_1 modulo b_2.

- When the above steps are implemented in the form of an algorithm, we have the **Extended Euclid's Algorithm**

5.6.6: Two Examples That Illustrate the Extended Euclid's Algorithm

Let's find the multiplicative inverse of 32 modulo 17:

```
gcd( 32, 17 )
    = gcd( 17, 15 )     | residue  15  = 1x32 - 1x17
    = gcd( 15, 2 )      | residue   2  = 1x17 - 1x15
                        |              = 1x17 - 1x(1x32 - 1x17)
                        |              = (-1)x32 + 2x17
    = gcd(  2, 1 )      | residue   1  = 1x15 - 7x2
                        |              = 1x(1x32 - 1x17)
                        |                   - 7x( (-1)x32 + 2x17 )
                        |              = 8x32 - 15x17
```

Therefore the multiplicative inverse of 32 modulo 17 is 8.

Let's now find the multiplicative inverse of 17 modulo 32:

```
gcd( 17, 32 )
    = gcd( 32, 17 )     | residue  17  = 1x17 + 0x32
    = gcd( 17, 15 )     | residue  15  = -1x17 + 1x32
    = gcd( 15, 2  )     | residue   2  = 1x17 - 1x15
                        |              = 1x17 - 1x( 1x32 - 1x17 )
                        |              = 2x17 - 1x32
    = gcd(  2, 1  )     | residue   1  = 15 - 7x2
                        |              = (1x32 - 1x17)
                        |                    - 7x(2x17 - 1x32)
                        |              = (-15)x17 + 8x32
                        |              = 17x17 + 8x32
                        |                (since the additive
                        |                 inverse of 15 is 17 mod 32)
```

Therefore the multiplicative inverse of 17 modulo 32 is 17.

5.7: THE EXTENDED EUCLID'S ALGORITHM IN PERL AND PYTHON

- So our quest for finding the multiplicative inverse (MI) of a number *num* modulo *mod* boils down to expressing the residues at each step of Euclid's recursion as a linear sum of *num* and *mod*, and, when the recursion terminates, taking for MI the coefficient of *num* in the final linear summation.

- As we step through the recursion called for by Euclid's algorithm, the originally supplied values for *num* and *mod* become modified as shown earlier. So let's use NUM to refer to the originally supplied value for *num* and MOD to refer to the originally supplied value for *mod*.

- Let x represent the coefficient of NUM and y the coefficient of MOD in our linear summation expressions for the residue at each step in the recursion. So our goal is to express the residue at each step in the form

$$residue \quad = \quad x * NUM \quad + \quad y * MOD \tag{8}$$

And then, when the *residue* is 1, to take the value of x as the multiplicative inverse of NUM modulo MOD, assuming, of course, the MI exists.

- What is interesting is that if you stare at the two examples shown in the previous section long enough (and, play with more examples like that), you will make the discovery that, as the Euclid's recursion proceeds, the new values of x and y can be computed directly from their current values and their previous values (which we will denote x_{old} and y_{old}) by the formulas:

$$\begin{aligned} x &<= x_{old} - q * x \\ y &<= y_{old} - q * y \end{aligned}$$

where q is the integer quotient obtained by dividing *num* by *mod*. To establish this fact, the following table illustrates again the second of the two examples shown in the previous section. This is the example for calculating $gcd(17, 32)$ where we are interested in finding the MI of 17 modulo 32:

Row	q = num//mod	num	mod	x	y
A. Initialization				1	0
B.		17	32	0	1
C. gcd(17, 32)					
D. residue = 17	17//32 = 0	32	17	1	0
E. gcd(32, 17)					
F. residue = 15	32//17 = 1	17	15	-1	1
G. gcd(17, 15)					

```
                              |                |       |     |     |     |
H.          residue = 2       | 17//15 =  1    |  15   |  2  |  2  |  -1 |
                              |                |       |     |     |     |
I.   gcd(15, 2)               |                |       |     |     |     |
                              |                |       |     |     |     |
J.          residue = 1       | 15//2 =  7     |   2   |  1  | -15 |  8  |
                              |                |       |     |     |     |
-------------------------------------------------------------------------
```

- Note the following rules for constructing the above table:
  - Rows A and B of the table are for initialization. We set x_{old} and y_{old} to 1 and 0, respectively, and their current values to 0 and 1. At this point, *num* is 17 and *mod* 32.

  - Note that the first thing we do in each new row is to calculate the quotient obtained by dividing the current *num* by the current *mod*. Only after that we update the values of *num* and *mod* in that row according to Euclid's recursion. For example, when we calculate q in row F, the current *num* is 32 and the current *mod* 17. Since the integer quotient obtained when you divide 32 by 17 is 1, the value of q in this row is 1. Having obtained the residue, we now invoke Euclid's recursion, which causes *num* to become 17 and *mod* to become 15 in row F.

  - We update the values of x on the basis of its current value and its previous value and the current value of the quotient q. For example, when we calculate the value of x in row J, the current value for x at that point is the one shown in row H, which is 2, and the previous value for x is shown in row F,

which is -1. Since the current value for the quotient q is 7, we obtain the new value of x in row J by $-1-7*2 = -15$. This is according to the update formula for the x coefficients: $x = x_{old} - q \times x$.

– The same goes for the variable y. It is updated in the same manner through the formula $y = y_{old} - q \times y$.

- Shown below is a Python implementation of the table construction presented above. The script shown is called with two command-line integer arguments. The **first** argument is the number whose MI you want to calculate and the **second** argument the modulus. As you'd expect, the MI exists only when $gcd(first, second) = 1$. When the MI does not exist, it prints out a "NO MI" message, followed by printing out the value of the gcd.

```
#!/usr/bin/env python

## FindMI.py

import sys

if len(sys.argv) != 3:
    sys.stderr.write("Usage: %s   <integer>   <modulus>\n" % sys.argv[0])
    sys.exit(1)

NUM, MOD = int(sys.argv[1]), int(sys.argv[2])

def MI(num, mod):
    '''
    This function uses ordinary integer arithmetic implementation of the
    Extended Euclid's Algorithm to find the MI of the first-arg integer
    vis-a-vis the second-arg integer.
    '''
    NUM = num; MOD = mod
    x, x_old = 0L, 1L
    y, y_old = 1L, 0L
```

```
    while mod:
        q = num // mod
        num, mod = mod, num % mod
        x, x_old = x_old - q * x, x
        y, y_old = y_old - q * y, y
    if num != 1:
        print("\nNO MI. However, the GCD of %d and %d is %u\n" % (NUM, MOD, num))
    else:
        MI = (x_old + MOD) % MOD
        print("\nMI of %d modulo %d is: %d\n" % (NUM, MOD, MI))

MI(NUM, MOD)
```

- When you invoke the above script by

```
FindMI.py 892347579824379987 89234759842347599
```

  it comes with the answer

```
MI of 892347579824379987 modulo 89234759842347599 is: 12596412217821807
```

- On the other hand, if you were to call

```
FindMI.py  16  32
```

  you will get the answer "NO MI. However, the GCD of 16 and 32 is 16."

- Shown on the next page is a Perl implementation of the same logic that was shown in the Python script above:

```
#!/usr/bin/env perl

##  FindMI.pl
##  Avi Kak

use strict;
use warnings;

die "\nUsage:   $0  <integer>  <integer>\n\n" unless @ARGV == 2;

die "At least one of your numbers is too large! Use FindMIWithBigInt.pl instead\n"
    if ($ARGV[0] > 0x7f_ff_ff_ff) or ($ARGV[1] > 0x7f_ff_ff_ff);

my ($NUM,$MOD) = @ARGV;

MI($NUM, $MOD);

##  This function uses ordinary integer arithmetic implementation of the
##  Extended Euclid's Algorithm to find the MI of the first-arg integer
##  vis-a-vis the second-arg integer.
sub MI {
    my ($num, $mod) = @_;
    my ($x, $x_old) = (0, 1);
    my ($y, $y_old) = (1, 0);
    while ($mod) {
        my $q = int($num / $mod);
        ($num, $mod) = ($mod, $num % $mod);
        ($x, $x_old) = ($x_old - $q * $x, $x);
        ($y, $y_old) = ($y_old - $q * $y, $y);
    }
    if ($num != 1) {
        print "\nNO MI. However, the GCD of $NUM and $MOD is $num\n\n"
    } else {
        my $MI = ($x_old + $MOD) % $MOD;
        print "\nMI of $NUM modulo $MOD is: $MI\n\n";
    }
}
```

- As was the case with our Perl implementation for the GCD algorithm, without the help of the `Math::BigInt` library, the script shown above will give correct results only when the numbers involved do not require more than 4 bytes for their representation. Shown below is a Perl implementation that uses `Math::BigInt`:

```
#!/usr/bin/env perl

##  FindMIWithBigInt.pl
##  Avi Kak

use strict;
use warnings;
use Math::BigInt;

die "\nUsage:   $0  <integer>  <integer>\n\n" unless @ARGV == 2;

my ($NUM,$MOD) = @ARGV;

$NUM = Math::BigInt->new("$NUM");
$MOD = Math::BigInt->new("$MOD");

MI($NUM, $MOD);

##  This function uses ordinary integer arithmetic implementation of the
##  Extended Euclid's Algorithm to find the MI of the first-arg integer
##  vis-a-vis the second-arg integer.
sub MI {
    my ($num, $mod) = @_;
    my ($x, $x_old) = (Math::BigInt->bzero(), Math::BigInt->bone());
    my ($y, $y_old) = (Math::BigInt->bone(), Math::BigInt->bzero());
    while ($mod->is_pos()) {
        my $q = $num->copy()->bdiv($mod);
        ($num, $mod) = ($mod, $num->copy()->bmod($mod));
        ($x, $x_old) = ($x_old->bsub( $q->bmul($x) ), $x);
        ($y, $y_old) = ($y_old->bsub( $q->bmul($y)), $y);
    }
    if ( ! $num->is_one() ) {
        print "\nNO MI. However, the GCD of $NUM and $MOD is $num\n\n"
    } else {
        my $MI = $x_old->badd( $MOD )->bmod( $MOD );
        print "\nMI of $NUM modulo $MOD is: $MI\n\n";
    }
}
```

- When you invoke the above script by

  ```
  FindMIWithBigInt.pl 892347579824379987 89234759842347599
  ```

  it comes back with the answer

  ```
  MI of 892347579824379987 modulo 89234759842347599 is: 12596412217821807
  ```

5.8: HOMEWORK PROBLEMS

1. What do we get from the following mod operations:

$$\begin{aligned} 2 \bmod 7 &= ? \\ 8 \bmod 7 &= ? \\ -1 \bmod 8 &= ? \\ -19 \bmod 17 &= ? \end{aligned}$$

Don't forget that, when the modulus is n, the result of a mod operation must be an integer between 0 and $n-1$, both ends inclusive, regardless of what quotient you have to use for the division. [When the dividend, such as the number -19 above, is negative, you'll have no choice but to use a negative quotient in order for the remainder to be between 0 and $n-1$, both ends inclusive.]

2. What is the difference between the notation

$$a \bmod n$$

and the notation

$$a \equiv b \pmod{n}$$

3. What is the notation for expressing that a is a divisor of b, that is when $b = m \times a$ for some integer m?

4. Consider the following equality:

$$(p \;+\; q)\; mod\; n \quad = \quad [\,(p\; mod\; n) \;+\; (q\; mod\; n)\,]\;\; mod\; n$$

Choose numbers for p, q, and n that show that the following version of the above is NOT correct:

$$(p \;+\; q)\; mod\; n \quad = \quad (p\; mod\; n) \;+\; (q\; mod\; n)$$

5. The notation Z_n stands for the set of residues. What does that mean?

6. How would you explain that Z_n is a commutative ring?

7. If I say that a number b in Z_n is the additive inverse of a number a in the same set, what does that say about $(a + b)\; mod\; n$?

8. If I say that a number b in Z_n is the multiplicative inverse of a number a in the same set, what does that say about $(a \times b)\; mod\; n$?

9. Is it possible for a number in Z_n to be its own additive inverse? Give an example.

10. Is it possible for a number in Z_n to be its own multiplicative inverse? Give an example.

11. Why is Z_n not an integral domain?

12. Why is Z_n not a finite field?

13. What are the asymmmetries between the modulo n addition and modulo n multiplication over Z_n?

14. Is it true that there exists an additive inverse for every number in Z_n regardless of the value of n?

15. Is it true that there exists a multiplicative inverse for every number in Z_n regardless of the value of n?

16. For any given n, what special property is satisfied by those numbers in Z_n that possess multiplicative inverses?

17. What is Euclid's algorithm for finding the GCD of two numbers?

18. How do you prove the correctness of Euclid's algorithm?

19. What is Bezout's identity for the GCD of two numbers?

20. How do we use Bezout's identity to find the multiplicative inverse of an integer in Z_p?

21. Find the multiplicative inverse of each nonzero element in Z_{11}.

22. **Programming Assignment:** Rewrite and extend the Python implementation of the *binary* GCD algorithm presented in Section 5.4.4 so that it incorporates the Bezout's Identity to yield multiplicative inverses. In other words, create a binary version of the multiplicative-inverse script of Section 5.7 that finds the answers by implementing the multiplications and division as bit shift operations.

23. **Programming Assignment:**

    All of the Python scripts shown in this lecture will work for arbitrary sized integers — simply because Python has the ability to create appropriate internal representations for arbitrary sized integers. On the other hand, when the size of an integer in Perl exceeds what can be stored as an unsigned int in 4 bytes, it creates an 8-byte floating point representation for the number. What that means is that while you can expect Python to keep on returning correct answers as the numbers get bigger, at some point the answers returned by Perl will start to be wrong. Your first task in this programming assignment is to see this effect for yourself by calling the Python and the Perl scripts with larger and larger integers.

    And your second task is to use Perl's `Math::BigInt` library to

modify the Perl scripts shown so that the answers returned are always correct for integers of any size.

24. **Programming Assignment:**

As you will see later, prime numbers play a critical role in many different types of algorithms important to computer security. A highly inefficient way to figure out whether an integer n is prime is to construct its set of remainders Z_n and to find out whether every element in this set, except of course the element 0, has a multiplicative inverse. Write a Python script that calls the MI script of Section 5.7 to find out whether all of the elements in the set Z_n for your choice of n possess multiplicative inverses. Your script should prompt the user for a value for n. Try your script for increasingly larger values of n — especially values with more than six decimal digits. For each n whose value you enter when prompted, your script should print out whether it is a prime or not.