

Discrete Structures 2

Lecture 3:

Number Theory

Part 3

Dr Abdullah Alharbi

Second Semester
(1447 AH – 2026 AD)

The lecture content is taken from Discrete Mathematics and Its Applications, 8th edition, by Kenneth H. Rosen. McGraw-Hill, 2019.

Lectures Contents:

- **Number Theory**:
  - Divisibility & Modular Arithmetic
  - Integer Representations & Algorithms
  - Primes & Greatest Common Divisors
  - **Solving Congruences**
  - **Applications of Congruences**

Number Theory:

- Greatest Common Divisor:
  - The Euclidean Algorithm
  - gcds as Linear Combinations:
    - Bézout's Theorem
- Solving Congruences:
  - Introduction
  - Linear Congruences:
    - Inverse of "a" modulo "m"
- Applications of Congruences:
  - Hashing Functions
  - Pseudorandom Numbers
  - Check Digits

The Euclidean Algorithm: Overview

- Computing the **greatest common divisor** of two integers directly from the prime factorizations of these integers is inefficient.
- The reason is that it is **time-consuming** to find prime factorizations.
- We will give a more efficient method of finding the **greatest common divisor**, called the Euclidean algorithm.
- This algorithm has been known since ancient times.
- It is named after the ancient Greek mathematician **Euclid**, who included a description of this algorithm in his book "The Elements".
- Before describing the Euclidean algorithm, we will show how it is used to find **gcd(91, 287)**.

The Euclidean Algorithm: Overview

- **First**, divide 287, the larger of the two integers, by 91, the smaller, to obtain: **287 = 91 · 3 + 14**
  - Any divisor of 91 and 287 must also be a divisor of **287 – 91 · 3 = 14**
  - Also, any divisor of 91 and 14 must also be a divisor of **287 = 91 · 3 + 14**
  - Hence, the greatest common divisor of 91 and 287 is the same as the greatest common divisor of 91 and 14.
    - This means that the problem of finding **gcd(91, 287)** has been reduced to the problem of finding **gcd(91, 14)**.
- **Next**, divide 91 by 14 to obtain **91 = 14 · 6 + 7**
  - Because any common divisor of 91 and 14 also divides **91 – 14 · 6 = 7** and any common divisor of 14 and 7 divides 91, it follows that **gcd(91, 14) = gcd(14, 7)**.
- **Continue** by dividing 14 by 7, to obtain **14 = 7 · 2**
- Because 7 divides 14, it follows that **gcd(14, 7) = 7**
  - Furthermore, because **gcd(287, 91) = gcd(91, 14) = gcd(14, 7) = 7**, the original problem has been solved.

The Euclidean Algorithm

- We now describe how the Euclidean algorithm works in generality.
- We will use successive divisions to reduce the problem of finding the greatest common divisor of two positive integers to the same problem with smaller integers, until one of the integers is **zero**.
- The **Euclidean algorithm** is based on the following result about greatest common divisors and the division algorithm.
- **Lemma 1**:

Let $a = bq + r$, where a, b, q, and r are integers. Then $\gcd(a, b) = \gcd(b, r)$.

The Euclidean Algorithm: Example

- **Example:** Find the greatest common divisor of **414** and **662** using the Euclidean algorithm.
- **Solution:** Successive uses of the **division algorithm** give:
  - **662** = **414** · 1 + **248**
  - **414** = **248** · 1 + **166**
  - **248** = **166** · 1 + **82**
  - **166** = **82** · 2 + **2**
  - **82** = **2** · 41 + **0**

The Euclidean Algorithm

- The Euclidean algorithm is expressed in pseudocode in **Algorithm 1**.

ALGORITHM 1 The Euclidean Algorithm.

procedure $gcd(a, b$: positive integers)
$x := a$
$y := b$
while $y \neq 0$
    $r := x$ **mod** y
    $x := y$
    $y := r$
return $x\{\gcd(a, b)$ is $x\}$

gcds as Linear Combinations

- The greatest common divisor of two integers "a" and "b" can be expressed in the form: **sa + tb**, where "s" and "t" are integers.
- In other words, **gcd(a, b)** can be expressed as a linear combination with integer coefficients of "**a**" and "**b**".
  - For example, **gcd(6, 14) = 2**, and **2 = (–2) · 6 + 1 · 14**
- We state this fact as **Theorem 6**.
- **Theorem 6**:

BÉZOUT'S THEOREM If a and b are positive integers, then there exist integers s and t such that $\gcd(a, b) = sa + tb$.

Bézout's Theorem

- **Definition 6**:

If a and b are positive integers, then integers s and t such that $\gcd(a, b) = sa + tb$ are called *Bézout coefficients* of a and b (after Étienne Bézout, a French mathematician of the eighteenth century). Also, the equation $\gcd(a, b) = sa + tb$ is called *Bézout's identity*.

- **Example**:

  Express **gcd(252, 198) = 18** as a *linear combination* of **252** and **198**

- This method illustrated above is a **two-pass** method.
- It first uses the **Euclidian algorithm** to find the **gcd**.
- Then, the method works **backwards** to express the **gcd** as a linear combination of the original two integers.

Bézout's Theorem

- **Solution:**
  - First use the **Euclidean algorithm** to show **gcd(252, 198) = 18**

i. $252 = 198 \cdot 1 + 54$

ii. $198 = 54 \cdot 3 + 36$

iii. $54 = 36 \cdot 1 + 18$

iv. $36 = 18 \cdot 2 + 0$

- Now working **backwards**, from **iii** and **ii** above, we can express **gcd(252, 198) = 18** as a **linear combination** of **54** and **36**.
  - $18 = 54 - 1 \cdot 36$
  - $36 = 198 - 3 \cdot 54$
- **Substituting** the **2nd equation** into the **1st** yields:
  - $18 = 54 - 1 \cdot (198 - 3 \cdot 54)$
  - $18 = 4 \cdot 54 - 1 \cdot 198$
- **Substituting "54 = 252 – 1 · 198" (from i))** yields:
  - $18 = 4 \cdot (252 - 1 \cdot 198) - 1 \cdot 198$
  - $18 = 4 \cdot 252 - 5 \cdot 198$

Consequences of Bézout's Theorem

- **Lemma 2**:

> If a, b, and c are positive integers such that $\gcd(a, b) = 1$ and $a \mid bc$, then $a \mid c$.

- **Lemma 3**:

> If p is a prime and $p \mid a_1 a_2 \cdots a_n$, where each a_i is an integer, then $p \mid a_i$ for some i.

Solving Congruences: Introduction

- Solving **linear congruences**, which have the form **$ax \equiv b \pmod{m}$**, is an essential task in the study of number theory and its applications, just as solving linear equations plays an important role in calculus and linear algebra.
- To solve linear congruences, we employ **inverses modulo "m"**.
- We explain how to work *backwards* through the steps of the **Euclidean algorithm** to find **inverses modulo "m"**.
- Once we have found an **inverse of "a" modulo "m"**, we solve the congruence **$ax \equiv b \pmod{m}$** by multiplying both sides of the congruence by this inverse.

Linear Congruences

- **Definition**: A congruence of the form $ax \equiv b \pmod{m}$, where m is a positive integer, a and b are integers, and x is a variable, is called a ***linear congruence***.
- How can we solve the linear congruence $ax \equiv b \pmod{m}$?
- The **solutions** to a linear congruence $ax \equiv b \pmod{m}$ are all integers x that satisfy the congruence.
- One method that we will describe uses an integer $\bar{a}$ such that:
  - $\bar{a}a \equiv 1 \pmod{m}$, *if such an integer exists*.
  - Such an integer $\bar{a}$ is said to be an **inverse of "a" modulo "m"**.
- **Theorem 1** guarantees that an **inverse of "a" modulo "m"** exists whenever "**a**" and "**m**" are **relatively prime**.
- **Definition**: An integer $\bar{a}$ such that $\bar{a}a \equiv 1 \pmod{m}$ is said to be an *inverse* of ***"a" modulo "m"***.

Inverse of "*a*" modulo "*m*"

- **Theorem 1**:

> If a and m are relatively prime integers and $m > 1$, then an inverse of a modulo m exists. Furthermore, this inverse is unique modulo m. (That is, there is a unique positive integer $\overline{a}$ less than m that is an inverse of a modulo m and every other inverse of a modulo m is congruent to $\overline{a}$ modulo m.)

- Using **inspection** to find an inverse of "a" modulo "m" is **easy** when "m" is small.
- To find this inverse, we look for a multiple of "a" that exceeds a multiple of "m" by 1.
- For example, to find an inverse of "**3 modulo 7**", we can find "**j · 3**" for "**j = 1, 2,…, 6,**" ***stopping*** when we find a **multiple of "3" that is one more than a multiple of "7"**.

Inverse of "*a*" **modulo** "*m*"

- We can design a more efficient algorithm than **brute-force** to find an inverse of "a" modulo "m" when "**gcd(a, m) = 1**" using the steps of the **Euclidean algorithm**.
- By reversing these steps as in the example in slide 10 & 11, we can find a **linear combination** "**sa + tm = 1**", where "**s**" and "**t**" are integers.
- Reducing both sides of this equation "**modulo m**" tells us that "**s**" is an **inverse of "a" modulo "m"**.
- We illustrate this procedure in the next example.

Finding Inverses

- The **Euclidean algorithm** and **Bézout coefficients** gives us a systematic approaches to finding inverses.
- **Example**: Find an inverse of 3 modulo 7.
- **Solution**: Because **gcd(3, 7) = 1**, by **Theorem 1**, *an inverse of 3 modulo 7 exists*.
  - Using the Euclidian algorithm: **7** = 2 · **3** + **1**
  - From this equation, we get **–2 · 3 + 1 · 7 = 1**,
    - See that **–2** and **1** are Bézout coefficients of **3** and **7**.
  - Hence, **–2** is an inverse of 3 modulo 7.
  - Also, every integer congruent to −2 modulo 7 is an inverse of 3 modulo 7, i.e., 5, −9, 12, etc.

Finding Inverses

Example: Find an inverse of 101 modulo 4620.

Solution: First use the Euclidian algorithm to show that **gcd(101, 4620) = 1**

$4620 = 45 \cdot 101 + 75$

$101 = 1 \cdot 75 + 26$

$75 = 2 \cdot 26 + 23$

$26 = 1 \cdot 23 + 3$

$23 = 7 \cdot 3 + 2$

$3 = 1 \cdot 2 + 1$

$2 = 2 \cdot 1$

Since the last nonzero remainder is 1, gcd(101, 4620) = 1

Working Backwards:

$1 = 3 - 1 \cdot 2$

$1 = 3 - 1 \cdot (23 - 7 \cdot 3) = -1 \cdot 23 + 8 \cdot 3$

$1 = -1 \cdot 23 + 8 \cdot (26 - 1 \cdot 23) = 8 \cdot 26 - 9 \cdot 23$

$1 = 8 \cdot 26 - 9 \cdot (75 - 2 \cdot 26) = 26 \cdot 26 - 9 \cdot 75$

$1 = 26 \cdot (101 - 1 \cdot 75) - 9 \cdot 75$

$= 26 \cdot 101 - 35 \cdot 75$

$1 = 26 \cdot 101 - 35 \cdot (4620 - 45 \cdot 101)$

$= -35 \cdot 4620 + 1601 \cdot 101$

Bézout coefficients : **– 35** and **1601**

1601 is an inverse of 101 modulo 4620

Using **Inverses** to **Solve Congruences**

- We can solve the congruence $ax \equiv b \pmod{m}$ by multiplying both sides by $\bar{a}$
- **Example**: What are the solutions of the congruence $3x \equiv 4 \pmod{7}$?
- **Solution**: We found that -2 is an inverse of 3 modulo 7 (slides 17)
- We multiply both sides of the congruence by -2 giving $-2 \cdot 3x \equiv -2 \cdot 4 \pmod{7}$
- Because $-6 \equiv 1 \pmod{7}$ and $-8 \equiv 6 \pmod{7}$, it follows that if x is a solution, then $x \equiv -8 \equiv 6 \pmod{7}$
- We need to determine if every x with $x \equiv 6 \pmod{7}$ is a solution.
- Assume that $x \equiv 6 \pmod{7}$
- By **Theorem 5** in "**Lecture 01: Part 1 (slide 20)**", it follows that $3x \equiv 3 \cdot 6 = 18 \equiv 4 \pmod{7}$ which shows that all such x satisfy the congruence
- The solutions are the integers x such that $x \equiv 6 \pmod{7}$, namely, $6, 13, 20 \ldots$ and $-1, -8, -15, \ldots$

Applications of Congruences

- **Congruences** have many applications to discrete mathematics, computer science, and many other disciplines.
- We will introduce three applications in this lecture:
  - The use of congruences to assign memory locations to computer files:
    - Hashing functions
  - The generation of pseudorandom numbers.
  - The generation of check digits.

Hashing Functions

- The central computer at an insurance company maintains records for each of its customers.
- How can memory locations be assigned so that customer records can be retrieved quickly?
- The **solution** to this problem is to use a suitably chosen **hashing function**.
- Records are identified using a **key**, which uniquely identifies each customer's records.
- For instance, customer records are often identified using the **Social Security number** of the customer as the **key**.
- A **hashing function** ***h*** assigns memory location ***h(k)*** to the record that has ***k*** as its **key**.

Hashing Functions

- In practice, many different hashing functions are used.
- One of the most common is the function "***h*(*k*) = *k* mod *m***" where ***m*** is the number of available memory locations.
- Hashing functions should be easily evaluated so that files can be quickly located.
- The hashing function "***h*(*k*) = *k* mod *m***" meets this requirement;
  - to find "***h*(*k*)**", we need only compute the remainder when "***k***" is divided by "***m***".
- Furthermore, the hashing function should be **onto**, so that all memory locations are possible.
  - The function "**h(k) = *k* mod *m***" also satisfies this property.

Hashing Functions

- **Example 1**: Find the memory locations assigned by the hashing function "**h(k) = k mod 111**" to the records of customers with Social Security numbers **064212848** and **037149212**.
- **Solution**:
- The record of the customer with Social Security number **064212848** is assigned to memory location **14**, because
- *h(**064212848**) = 064212848 mod 111 = **14***
- Similarly, because
- *h(**037149212**) = 037149212 mod 111 = **65***
- the record of the customer with Social Security number **037149212** is assigned to memory location **65**.

Hashing Functions: Collision

- Because a hashing function is **not one-to-one** (because there are more possible keys than memory locations), more than one file may be assigned to a memory location.
- When this happens, we say that a **collision** occurs.
- One way to resolve a collision is *to assign the first free location following the occupied memory location assigned by the hashing function*.
- **Example** 2: After making the assignments of records to memory locations in **Example 1**, assign a memory location to the record of the customer with Social Security number **107405723**.

Hashing Functions: Collision

- **Solution:** First note that the hashing function "**h(k) = k mod 111**" maps the Social Security number **107405723** to location **14**, because
- *h(107405723) = 107405723 mod 111 =* **14**
- However, this location is already occupied (by the file of the customer with Social Security number **064212848**).
- But, because memory location **15**, the first location following memory location **14**, is **free**, we assign the record of the customer with Social Security number **107405723** to this location.
- In **Example 1** we used a **linear probing function**, namely, "**h(k, i) = h(k) + i mod m**", to look for the first free memory location, where "**i**" runs from **0** to **m – 1**.
- There are many other ways to resolve collisions.

Pseudorandom Numbers

- Randomly chosen numbers are often needed for **computer simulations**.
- Different methods have been devised for generating numbers that have properties of randomly chosen numbers.
- Because numbers generated by systematic methods are not truly random, they are called **pseudorandom numbers**.
- The most commonly used procedure for generating pseudorandom numbers is the **linear congruential method**.
- We choose **four** integers: the **modulus** "**m**", **multiplier** "**a**", **increment** "**c**", and **seed** $\mathbf{x_0}$, with $2 \leq a < m$, $0 \leq c < m$, and $0 \leq x_0 < m$
- We generate a sequence of pseudorandom numbers $\{x_n\}$, with "$0 \leq x_n < m$" for all "n", by successively using the recursively defined function:
- $x_{n+1} = (ax_n + c) \bmod m$

Pseudorandom Numbers

- Many computer experiments require the generation of pseudorandom numbers **between 0 and 1**.
- To generate such numbers:
  - We divide numbers generated with a **linear congruential generator** by the **modulus**: that is, we use the numbers "x_n / m"
- **Example 3**: Find the sequence of pseudorandom numbers generated by the linear congruential method with modulus "**m = 9**", multiplier "**a = 7**", increment "**c = 4**", and seed "$\mathbf{x_0 = 3}$"

Solution: We compute the terms of this sequence by successively using the recursively defined function $x_{n+1} = (7x_n + 4) \bmod 9$, beginning by inserting the seed $x_0 = 3$ to find x_1. We find that

$$
\begin{aligned}
x_1 &= 7x_0 + 4 \bmod 9 = 7 \cdot 3 + 4 \bmod 9 = 25 \bmod 9 = 7,\\
x_2 &= 7x_1 + 4 \bmod 9 = 7 \cdot 7 + 4 \bmod 9 = 53 \bmod 9 = 8,\\
x_3 &= 7x_2 + 4 \bmod 9 = 7 \cdot 8 + 4 \bmod 9 = 60 \bmod 9 = 6,\\
x_4 &= 7x_3 + 4 \bmod 9 = 7 \cdot 6 + 4 \bmod 9 = 46 \bmod 9 = 1,\\
x_5 &= 7x_4 + 4 \bmod 9 = 7 \cdot 1 + 4 \bmod 9 = 11 \bmod 9 = 2,\\
x_6 &= 7x_5 + 4 \bmod 9 = 7 \cdot 2 + 4 \bmod 9 = 18 \bmod 9 = 0,\\
x_7 &= 7x_6 + 4 \bmod 9 = 7 \cdot 0 + 4 \bmod 9 = \ \ 4 \bmod 9 = 4,\\
x_8 &= 7x_7 + 4 \bmod 9 = 7 \cdot 4 + 4 \bmod 9 = 32 \bmod 9 = 5,\\
x_9 &= 7x_8 + 4 \bmod 9 = 7 \cdot 5 + 4 \bmod 9 = 39 \bmod 9 = 3.
\end{aligned}
$$

Because $x_9 = x_0$ and because each term depends only on the previous term, we see that the sequence

3, 7, 8, 6, 1, 2, 0, 4, 5, 3, 7, 8, 6, 1, 2, 0, 4, 5, 3, …

is generated. This sequence contains nine different numbers before repeating.

Pseudorandom Numbers

- Most computers do use **linear congruential generators** to generate pseudorandom numbers.
- Often, a **linear congruential generator** with increment "**c = 0**" is used.
- Such a generator is called a **pure multiplicative generator**.
- For example, the pure multiplicative generator with **modulus** "$\mathbf{2^{31} - 1}$" and **multiplier** "**75 = 16,807**" is widely used.
- With these values, it can be shown that "$\mathbf{2^{31} - 2}$" numbers are generated before repetition begins.

Pseudorandom Numbers

- Pseudorandom numbers generated by **linear congruential generators** have long been used for many tasks.
- **Unfortunately**, it has been shown that sequences of pseudorandom numbers generated in this way do not share some important statistical properties that true random numbers have.
- Because of this, it is not advisable to use them for some tasks, such as large simulations.
- For such sensitive tasks, other methods are used to produce sequences of pseudorandom numbers, either using some sort of algorithm or **sampling numbers** arising from a random physical phenomenon.

Check Digits

- **Congruences** are used to check for errors in digit strings.
- A common technique for detecting errors in such strings is to add an extra digit at the end of the string.
- This "**final digit**", or "**check digit**", is calculated using a particular function.
- Then, to determine whether a digit string is **correct**, a check is made to see whether this final digit has the **correct value**.
- **Check bits** computed using **congruences** are used extensively to verify the correctness of various kinds of identification numbers.
  - Universal Product Codes
  - International Standard Book Numbers
  - Money Order Numbers
  - Airline Ticket Numbers
  - Identification Numbers for Periodicals
  - Bank Account Numbers
  - Drivers License Numbers
  - Credit Card Numbers

Check Digits: Parity Check Bits

- **Digital information** is represented by **bit string**, split into blocks of a specified size.
- Before each block is stored or transmitted, **an extra bit**, called a **parity check** bit, can be appended to each block.
- The **parity check bit x_{n+1}** for the bit string $x_1x_2\ldots x_n$ is defined by:

$$x_{n+1} = x_1 + x_2 + \cdots + x_n \bmod 2$$

- It follows that x_{n+1} is:
  - **0** if there are an **even number of 1 bits** in the block of "n" bits
  - **1** if there are an **odd number of 1 bits** in the block of "n" bits.
- When we examine a string that includes a parity check bit, we know that there is an **error** in it if the **parity check** bit is **wrong**.

Check Digits: Parity Check Bits

- **Example 4**: Suppose we receive in a transmission the bit strings **01100101** and **11010110**, each ending with a parity check bit. Should we accept these bit strings as correct?
- **Solution**:
- Before accepting these strings as correct, we examine their parity check bits.
- The parity check bit of the first string is 1.
- Because $0 + 1 + 1 + 0 + 0 + 1 + 0 \equiv 1 \pmod 2$, the parity check bit is **correct**.
- The parity check bit of the second string is 0.
- We find that $1 + 1 + 0 + 1 + 0 + 1 + 1 \equiv 1 \pmod 2$, so the parity check is **incorrect**.
- We conclude that the first string may have been transmitted correctly and we know for certain that the second string was transmitted incorrectly.
- We accept the first string as correct (even though it still may contain an even number of errors), but we reject the second string.

Any Questions?

- Have a look at the similar "Extra Examples" on the textbook's website.
- Keep reviewing this lecture *regularly, as well as other lectures*.
- Have a look at the *exercises* at the end of the sections we covered in this lecture in Chapter 4 and do your best with those ones similar to what we presented in this lecture.
  - You should get more *examples* and *exercises* in the *tutorials*.
- Ask *me or ask your peers* if you have any questions.
  - Please, do not wait until the end of the semester or even right before exams.

All the best!