

Mat 2345 Chapter Two

Algorithms, Complexity, Integers

Fall 2007

Chapter Two Overview

- ▶ Algorithms
- ▶ Complexity of Algorithms
- ▶ The Integers and Division
- ▶ Integers and Algorithms
- ▶ Applications of Number Theory

Section 2.1 – Algorithms

- ▶ **Algorithm**: a finite set of unambiguous instructions for performing a computation or for solving a problem.
- ▶ Examples:
 - ▶ Shampoo Instructions: Lather, Rinse, Repeat
 - ▶ Recipe for making Italian Beef: Place beef roast, 1 pkg Au Jus dry gravy mix, 1 pkg dry Italian dressing mix, and 1 C water in slow cooker; cook all day or over night; shred beef and serve with French Bread or rolls
 - ▶ The instructions that come with a sewing pattern
 - ▶ The instructions of a model airplane or rocket kit
 - ▶ INSERT DISK AND PRESS ANY KEY TO CONTINUE

Properties of Algorithms

- ▶ **Input** - an algorithm usually has input from a specified set
- ▶ **Output** - the solution to the problem, also from a specified set
- ▶ **Definiteness** - steps of an algorithm must be defined precisely
- ▶ **Correctness** - an algorithm must produce the correct values for each of the input values

Properties of Algorithms – Continued

- ▶ **Finiteness** - an algorithm must produce the desired output after a finite (but perhaps large) number of steps for any input in the set
- ▶ **Effectiveness** - it must be possible to perform each step of an algorithm exactly and in a finite amount of time
- ▶ **Generality** - an algorithm should be applicable for all problems of the desired form, not just for a particular set of input values

Finding the Max in a Finite Sequence – Pseudocode

```
procedure max(a1, a2, ..., an : integers)
max := a1
for i := 2 to n
    if max < ai then max := ai
{max is the largest element}
```

Finding a Finite Sequence Max— Implementation

```
template <class T>
T Max (const vector<T> & L, const T & target){
// PRE: L not empty, target initialized
//     type T is comparable
// POST: returns the largest value in the
//       vector L

T mymax = L[1];

for (int i = 2; i < L.size; i++)
    if (mymax < L[i])
        mymax = L[i];

return mymax;
}
```

Search Algorithms

- ▶ The problem: locating a particular element, the **target**, in a list
- ▶ Distinguish between **unordered** and **ordered (sorted)** lists
- ▶ Two primary algorithms
 - ▶ **Linear or Sequential Search** — look at each item in the list, first to last, comparing them to target until target is found or reach end of list
 - ▶ **Binary Search** — (**only** used on **ordered** lists) — compare target to middle element; discard low or high half of list and repeat on remaining half of list until found or list is empty

The Linear Search Algorithm

```
procedure LinearSearch
(x: integer,
 a1, a2, ..., an: distinct integers)

i := 1

while (i <= n and x != ai)
    i := i + 1

if i <= n
    then location := i
    else location := 0

{ location is the subscript of term that
  equals x, or is 0 if x is not found }
```

Notice it doesn't matter if the list is ordered or not, this algorithm will still work

The Binary Search Algorithm

```
procedure BinarySearch
(x: integer,
 a1, a2, ..., an: increasing integers)

i := 1 {left endpoint of search interval}
j := n {right endpoint of search interval}

while i < j begin
    m := floor[(i + j) / 2]
    if x > am
        then i := m + 1
    else j := m
end

if x = ai
    then location := i
    else location := 0

{ location is the subscript of term that
  equals x, or is 0 if x is not found }
```

Section 2.2 — Complexity of Algorithms

- ▶ When does an algorithm produce a satisfactory solution to a problem?
- ▶ How can we prove an algorithm always produces the correct answer? (seen in next chapter)
- ▶ How can we analyze the efficiency of an algorithm?
 - ▶ **Time Complexity** — One measure is the number of steps it performs, or the time it takes, to solve a problem when input values are of a specified size
 - ▶ **Space Complexity** — A second measure is the amount of computer memory required during execution of an implementation of the algorithm, when input values are of a specified size

Time Complexity

- ▶ It is obviously important to know whether an algorithm will produce an answer in milliseconds or time measured in years.
- ▶ Time complexity can be described in terms of the number of operations required instead of actual computer time because of the difference in time needed for different computers to perform basic operations.
- ▶ It would be quite complicated to break down all operations to the basic bit operations that a computer uses
- ▶ Various machines, from personal computers to supercomputers, perform basic bit operations at rates which differ by as much as 1,000 times or more

Space Complexity

- ▶ It is obviously important to determine whether an algorithm will require more memory than we have available
- ▶ Space complexity can be described in terms of the amount of memory necessary to store one element \times the size of input, plus additional storage required by the algorithm, and often in terms of the size of input and its storage requirements
- ▶ Considerations of space complexity are tied to the particular data structures used to implement the algorithm

Analyzing Time Complexity

We can count comparisons, data movement, arithmetic operations, or other types of "steps." It just depends upon the problem and what's important to us

There are three types of analysis:

- ▶ **Best Case** — the run-time conditions are the best we can ever get; for example, the numbers are already sorted, or there is only one value in the list so it is the max. Doesn't give us very much information about the algorithm besides a lower-bound on execution time.
- ▶ **Average Case** — can be very difficult to determine, and cannot predict behavior for bad cases
- ▶ **Worst Case** — most commonly used (at undergraduate level); it gives us an upper-bound on execution time, but can be overly pessimistic

Analyzing the Search Algorithms

- ▶ Linear Search
- ▶ Binary Search (for simplicity, assume there are $n = 2^k$ elements in the input list)

Commonly Used Complexity Terminology

Complexity	Terminology
$O(1)$, $O(c)$	Constant complexity
$O(\log n)$	Logarithmic complexity
$O(n)$	Linear complexity
$O(n \log n)$	$n \log(n)$ complexity
$O(n^b)$	Polynomial complexity
$O(b^n)$, where $b > 1$	Exponential complexity
$O(n!)$	Factorial complexity

Classes of Problems

- ▶ Problems for which answers can be found using a computer are called **solvable**
- ▶ Problems which are not solvable by computer are called **unsolvable**

One (famous) example of an unsolvable problem:
The **Halting Problem** — can one program determine whether another arbitrary program will halt when executed with a specified input? (The answer is no.)

Tractable vs Intractable Problems

- ▶ A solvable problem is called **tractable** if there exists an algorithm with polynomial worst-case complexity to solve it.
- ▶ Even if a problem is tractable, there's no guarantee it can be solved in a reasonable amount of time for even relatively small input values.
- ▶ Most algorithms in use have polynomial complexities of degree 4 or less.

Tractable vs Intractable Problems — Continued

- ▶ Solvable algorithms with worst-case time complexities that exceed polynomial times are called **intractable**
- ▶ Usually, but not always, an extremely large amount of time is required to solve the problem for the worst cases of even small input values.
- ▶ In a few instances, an exponential or worse algorithm may be able to solve problems of reasonable size in sufficient time to be useful

Further Algorithm Classifications

- ▶ Tractable algorithms are said to belong to **class P** — polynomial time algorithms.
- ▶ It is commonly believed that many solvable problems have no polynomial time algorithm to solve them, but that **given** a possible solution, it can be **checked** in polynomial time.
- ▶ Problems for which a solution can be checked in polynomial time belong to the **class NP**
- ▶ NP stands for **Non-deterministic polynomial** time — we **guess** an answer then check it in **polynomial** time.

NP-Complete Problem Class

Another important class of problems, called **NP-Complete** problems, are problems in the class NP which have the property that:

If **any** of the problems in the **NP-Complete** class can be solved in polynomial time, then **all** of them can be

No one has been able to find such an algorithm.

It is suspected that no one ever will.

Section 2.3 — The Integers and Division

- ▶ Integers and their properties belong to a branch of Mathematics called **Number Theory**
- ▶ If a and b are integers, with $a \neq 0$, we say that **a divides b** if there is an integer c such that $b = ac$.

Notation: $a \mid b$, a divides b
 a is a **factor** of b ; b is a **multiple** of a
 $a \nmid b$, a does not divide b

Theorem. Let a , b and c be integers. Then:

The sum of multiples is a multiple:

if $a \mid b$ and $a \mid c$, then $a \mid (b + c)$

If an integer divides a factor, then it divides the product:

if $a \mid b$, then $a \mid bc$ for all integers c

Divisibility is transitive:

if $a \mid b$ and $b \mid c$, then $a \mid c$

Prime Numbers

- ▶ A positive integer $p > 1$ is called **prime** if the only positive factors of p are 1 and p .
- ▶ A positive integer that is greater than 1 and is not prime is called **composite**.
- ▶ **The Fundamental Theorem of Arithmetic.** Every positive integer can be written uniquely as the product of primes, where the prime factors are written in order of increasing size.

Showing Primality

Theorem. If n is a composite integer, then n has a prime divisor less than or equal to \sqrt{n} .

Show 103 is prime, using the above theorem and the fact $\sqrt{103} < 11$.

How to factor: attempt to divide by known primes beginning with 2, 3, 5, ...

Division of Integers

► **Theorem. The "Division Algorithm".**

Let a be an integer, d be a positive integer. Then there are unique integers q and r , with $0 \leq r < d$, such that $a = dq + r$.

► In the equality given in the division algorithm:

d is called the **divisor**,

a is called the **dividend**,

q is called the **quotient**, and

r is called the **remainder**

Relatively Prime Integer Pairs

► Let a and b be integers, not both zero. The largest integer d such that $d \mid a$ and $d \mid b$ is called the **greatest common divisor**, denoted $\gcd(a, b)$.

Find the greatest common divisors of the following integer pairs:

$\gcd(12, 39)$:

$\gcd(23, 103)$:

$\gcd(8, 9)$:

$\gcd(28, 42)$:

► The integers a and b are **relatively prime** if their greatest common divisor is 1.

Pairwise Relatively Prime Integers

► The integers a_1, a_2, \dots, a_n are **pairwise relatively prime** if $\gcd(a_i, a_j) = 1$ whenever $1 \leq i < j \leq n$.

Are the following sets of integers pairwise relatively prime?

► 22, 28, and 31

► $(2^3 \cdot 5 \cdot 11)$, $(3^3 \cdot 7)$, and $(13 \cdot 23^5)$

► $(2^3 \cdot 5 \cdot 11)$, $(3^2 \cdot 11^2)$, and 29

Least Common Multiple

The **Least Common Multiple** of the positive integers a and b , denoted $\text{lcm}(a, b)$, is the smallest positive integer that is divisible by both a and b .

What is the least common multiple of each of the following pairs?

► $\text{lcm}(22, 28) =$

► $\text{lcm}(2^3 \cdot 5 \cdot 11, 3^3 \cdot 7) =$

► $\text{lcm}(13 \cdot 23^5, 13^2) =$

► $\text{lcm}(2^3 \cdot 5 \cdot 11, 3^2 \cdot 11^2) =$

Product of gcd and lcm

Theorem. Let a and b be positive integers. Then:

$$ab = \gcd(a, b) \cdot \text{lcm}(a, b)$$

Consider:

► $600 = 2^3 \cdot 3 \cdot 5^2$ and $56250 = 2 \cdot 3^2 \cdot 5^5$

► $\text{lcm}(600, 56250) = 2^3 \cdot 3^2 \cdot 5^5$

► $\gcd(600, 56250) = 2 \cdot 3 \cdot 5^2$

► $\text{product}(600, 56250) = 2^4 \cdot 3^3 \cdot 5^7$

► $\text{product}(\text{lcm}(600, 56250), \gcd(600, 56250)) = 2^4 \cdot 3^3 \cdot 5^7$

Why are the last two products equivalent?

Modular Arithmetic

- ▶ Let a be an integer and m be a positive integer. We denote by $(a \bmod m)$ the **remainder** when a is divided by m .

From this definition, it follows that:

$$\text{if } (a \bmod m) = r, \text{ then } a = qm + r \text{ and } 0 \leq r < m.$$

- ▶ If a and b are integers, and m is a positive integer, then a is **congruent to b modulo m** if m divides $a - b$. This is denoted by: $a \equiv b \pmod{m}$

Note: $a \bmod m$ and $b \bmod m$ will yield the same remainder.

Consider: $(17 - 5) \bmod 6 = 12 \bmod 6 = 0$

Also: $17 \bmod 6 = 5$ and $5 \bmod 6 = 5$

Thus: $17 \equiv 5 \pmod{6}$

More on Congruency

- ▶ **Theorem.** Let m be a positive integer. The integers a and b are congruent modulo m if and only if there is an integer k such that $a = b + km$.

- ▶ **Theorem.** Let m be a positive integer. If $a \equiv b \pmod{m}$ and $c \equiv d \pmod{m}$, then

$$a + c \equiv (b + d) \pmod{m}$$

and

$$ac \equiv bd \pmod{m}$$

Consider: $7 \equiv 2 \pmod{5}$ and $11 \equiv 1 \pmod{5}$

Thus: $7 + 11 = 18 \equiv (2+1) \pmod{5} = 3 \pmod{5}$

And: $7 \cdot 11 = 77 \equiv (2 \cdot 1) \pmod{5} = 2 \pmod{5}$

Section 2.4 – Integers and Algorithms

- ▶ **Euclidean Algorithm:** an efficient method of finding the greatest common divisor, rather than factoring both numbers.
- ▶ An example of how it works: Find $\text{gcd}(91, 287)$
 1. Divide the larger number by the smaller one:
 $287 / 91 = 3 \text{ R } 14$, so
 $287 = 91(3) + 14$
 2. 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 $287 = 91(3) + 14$
 3. Thus, $\text{gcd}(91, 287) = \text{gcd}(14, 91)$; so divide 91 by 14
 $91 = 14(6) + 7$
 4. Same argument applies, so find $\text{gcd}(14, 7)$
 5. Hence, $\text{gcd}(91, 287) = \text{gcd}(14, 91) = \text{gcd}(7, 14) = 7$

Algorithm to Find $\text{gcd}()$

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

The Euclidean Algorithm

```
function gcd(a, b: positive integers)
  x ← a
  y ← b
  while (y != 0) {
    r ← x mod y
    x ← y
    y ← r
  } // end of loop to find gcd
  return x //the last non-zero remainder
} // end of gcd function
```

Find: $\text{gcd}(414, 662) =$

Integer Representations

- ▶ **Theorem.** Let b be a positive integer greater than 1. Then if n is a positive integer, it can be expressed uniquely in the form:

$$n = a_k b^k + a_{k-1} b^{k-1} + \dots + a_1 b + a_0$$

where k is a non-negative integer, a_0, a_1, \dots, a_k are non-negative integers less than b , and $a_k \neq 0$

- ▶ The representation of n given in this Theorem is called the **base b expansion of n** , denoted by $(a_k a_{k-1} \dots a_1 a_0)_b$
- ▶ Example I (octal): $(734)_8 = 7(8^2) + 3(8^1) + 4(8^0) = 476_{10}$
- ▶ Example II (binary): $1011001 = 2^6 + 2^4 + 2^3 + 2^0 = 89_{10}$

Hexadecimal – Base 16

- ▶ Hexadecimal or Base 16 digits are 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A (10), B (11), C (12), D (13), E (14), and F (15)

- ▶ Given 4 bits, we can represent 16 different values, 0 - F:

0000	0001	0010	0011
0100	0101	0110	0111
1000	1001	1010	1011
1100	1101	1110	1111

- ▶ Since one byte is 8 bits, we can represent a byte of information with two hexadecimal digits; so $01011101_2 = 5D_{16}$

- ▶ Example III:

$$(2AE0B)_{16} =$$

$$2(16^4) + 10(16^3) + 14(16^2) + 0(16^1) + 11(16^0) = 175,627_{10}$$

Conversion from Base 10

- ▶ Process to convert n_{10} to base b
 1. Divide n by b to obtain a quotient and remainder:

$$n = bq_0 + a_0, 0 \leq a_0 < b$$
 This remainder, a_0 , is the rightmost digit in the base b expansion of n .
 2. Divide q_0 by b : $q_0 = bq_1 + a_1, 0 \leq a_1 < b$
 This remainder, a_1 , is the second digit from the right-hand side in the base b expansion of n .
 3. Continue this process, successively dividing the quotients by b , obtaining additional base b digits as the remainders.
 4. The process terminates when we obtain a quotient equal to zero

Conversion Algorithm

Constructing Base b Expansions

```

procedure base_b_expansion (n: positive integer){
  q ← n
  k ← 0

  while (q != 0) {
    a[k] ← q mod b
    q ← floor(q / b)
    k ← k + 1
  } // end conversion loop

  return a
} // end expansion
    
```

Conversion Practice

- ▶ Find the base 8 expansion of $(532)_{10}$
- ▶ Find the base 2 expansion of $(532)_{10}$
- ▶ Find the base 16 expansion of $(532)_{10}$

Arithmetic Operations — Addition

- ▶ Addition in various bases is accomplished in a manner similar to base 10 addition

binary	octal	hex
$\begin{array}{r} 101100 \\ + 011010 \\ \hline \end{array}$	$\begin{array}{r} 7340 \\ + 521 \\ \hline \end{array}$	$\begin{array}{r} 29AC \\ + A131 \\ \hline \end{array}$

Arithmetic Operations — Multiplication

- ▶ Multiplication in various bases is accomplished in a manner similar to base 10 multiplication

binary	octal	hex
$\begin{array}{r} 101100 \\ \times 011010 \\ \hline \end{array}$	$\begin{array}{r} 7340 \\ \times 521 \\ \hline \end{array}$	$\begin{array}{r} 29AC \\ \times A131 \\ \hline \end{array}$

Representing Values in a Compute

Unsigned Integers

- ▶ non-negative integer representation
- ▶ used for such things as counting and memory addresses
- ▶ with k bits, exactly 2^k integers, ranging from 0 to $2^k - 1$ can be represented

Signed Integers

- ▶ If integers are stored in 8 bits, how many different bit patterns are there available to assign to various values?
- ▶ If we assign the bit pattern 0000000 to the value 0, how many are left for other values?
- ▶ There are different methods to deal with the "extra" bit pattern.

Signed Integer Representation Schemes

Use the high-order (leftmost) bit to represent the **sign** of the number: 0 for positive, 1 for negative. All positive numbers (beginning with a 0 bit) are simply evaluated as is. If the first bit is 1 (signifying a negative number):

1. **Signed Magnitude** — the other bits are evaluated to find the magnitude of the number (then make it negative).
2. **1's Complement** — flip (complement) the other bits before evaluating them to find the magnitude (then make it negative).
3. **2's Complement** — flip all the bits and add 00...01 before evaluating them to find the magnitude (then make it negative)

The following table is based upon a 4-bit representation. What happen when we add 1 and -1 in each representation?

bit pattern	Signed Magnitude	1's Complement	2's Complement
0000	0	0	0
0001	1	1	1
0010	2	2	2
0011	3	3	3
0100	4	4	4
0101	5	5	5
0110	6	6	6
0111	7	7	7
1000	-0	-7	-8
1001	-1	-6	-7
1010	-2	-5	-6
1011	-3	-4	-5
1100	-4	-3	-4
1101	-5	-2	-3
1110	-6	-1	-2
1111	-7	-0	-1

Russian Peasant Multiplication Algorithm

- ▶ An interesting multiplication algorithm based on the principle of doubling and halving — of the two numbers to be multiplied, one is halved and the other is doubled.
- ▶ Remainders when halving odd numbers are ignored.
- ▶ However, the numbers in the doubles column corresponding to the odd numbers in the halves column are chosen, and the rest are ignored.
- ▶ The sum of the selected numbers in the doubles column gives the product.

Examples

Find: 28 × 57	Find: 183 × 49																										
<table border="1"> <thead> <tr> <th>Halves</th> <th>Doubles</th> </tr> </thead> <tbody> <tr><td>28</td><td>57</td></tr> <tr><td>14</td><td>114</td></tr> <tr><td>7</td><td>228</td></tr> <tr><td>3</td><td>456</td></tr> <tr><td>1</td><td>912</td></tr> </tbody> </table> <p>228 + 456 + 912 = 1596</p>	Halves	Doubles	28	57	14	114	7	228	3	456	1	912	<table border="1"> <thead> <tr> <th>Halves</th> <th>Doubles</th> </tr> </thead> <tbody> <tr><td>49</td><td>183</td></tr> <tr><td>24</td><td>366</td></tr> <tr><td>12</td><td>732</td></tr> <tr><td>6</td><td>1464</td></tr> <tr><td>3</td><td>2928</td></tr> <tr><td>1</td><td>5856</td></tr> </tbody> </table> <p>183 + 2928 + 5856 = 8967</p>	Halves	Doubles	49	183	24	366	12	732	6	1464	3	2928	1	5856
Halves	Doubles																										
28	57																										
14	114																										
7	228																										
3	456																										
1	912																										
Halves	Doubles																										
49	183																										
24	366																										
12	732																										
6	1464																										
3	2928																										
1	5856																										

Why Does It Work?

Some observations:

- ▶ The method is compensatory in nature since with each subsequent step, a factor of 2 is moved from the number in the Halves column to the number in the Doubles column.
- ▶ The numbers in the Doubles column contain 1, 2, 4, 8, 16, ... groups of the given number in that column. (This shows a connection with the binary system of numeration)
- ▶ The retained rows correspond to the binary expansion of the Halved factor.

Distributive Property

Suppose we want to multiply 12 by 13:

$$\begin{array}{r}
 13 \\
 \times 12 \\
 \hline
 26 \\
 + 130 \\
 \hline
 156
 \end{array}$$

Notice that we add 2(13) and 10(13) to get the final answer.

This works because of the distributive property:

$$12 \cdot 13 = (2 + 10) \cdot 13 = 2 \cdot 13 + 10 \cdot 13$$

Of course, we can break 12 down into any factors we like and still get the right answer, so:

$$12 \cdot 13 = (4 + 8) \cdot 13 = 4 \cdot 13 + 8 \cdot 13 = 2^2 \cdot 13 + 2^3 \cdot 13$$

Doubling

▶ Thus, if we can multiply 13 by 2^2 and 2^3 , and add their products, we will be finished.

▶ Repeatedly doubling a number multiplies it by powers of two:

Number	Multiplications so far	Power of 2
13	13	2^0
26	$13 * 2$	2^1
52	$13 * 2 * 2$	2^2
104	$13 * 2 * 2 * 2$	2^3

▶ So $(2^2 * 13) + (2^3 * 13) = 52 + 104 = 156$

In Summary

Number Doubled	Multiplications so far	Power of 2	Number Halved	Division Problem	Remainder
13	13	2^0	12	$12/2 = 6$	0
26	$13 * 2$	2^1	6	$6/2 = 3$	0
52	$13 * 2 * 2$	2^2	3	$3/2 = 1$	1
104	$13 * 2 * 2 * 2$	2^3	1	$1/2 = 0$	1

Section 2.5 — Applied Number Theory

▶ **Theorem 1. (linear combination):** If $a, b \in \mathbb{Z}^+$, then $\exists s, t \in \mathbb{Z} \ni \gcd(a, b) = sa + tb$

▶ s & t can be found by working backward through the divisions of the Euclidean Algorithm

▶ Express $\gcd(154, 105)$ as linear combination of 252 and 198

Using the Euclidean Algorithm:

$$\begin{aligned} (2) \quad 154 &= 1(105) + 49 \\ (1) \quad 105 &= 2(49) + 7 \\ (0) \quad 49 &= 7(7) + 0 \end{aligned} \quad \text{so } \gcd(154, 105) = 7$$

Working Backwards:

$$\begin{aligned} \text{by (1)} \quad 7 &= 105 - 2(49) \\ \text{by (2)} \quad 49 &= 154 - 105 \\ \text{so } 7 &= 105 - 2(154 - 105) \\ &= 3(105) - 2(154) \end{aligned}$$

Linear Combination Example II

Find a linear combination of 252 and 198 which equals their gcd.

Using the Euclidean Algorithm:

$$\begin{aligned} (3) \quad 252 &= 1(198) + 54 \\ (2) \quad 198 &= 3(54) + 36 \\ (1) \quad 54 &= 1(36) + 18 \\ (0) \quad 36 &= 2(18) + 0 \end{aligned} \quad \text{so } \gcd(252, 198) = 18$$

Working Backwards:

$$\begin{aligned} \text{by (1)} \quad 18 &= 54 - 1(36) \\ \text{by (2)} \quad 36 &= 198 - 3(54) \\ \text{so } 18 &= 54 - 1(198 - 3(54)) \\ &= 4(54) - 198 \\ \text{by (3)} \quad 54 &= 252 - 1(198) \\ \text{so } 18 &= 4(252 - 198) - 198 \\ &= 4(252) - 5(198) \end{aligned}$$

Linear Combination Example III

Find a linear combination of 124 and 323 which equals their gcd.

Using the Euclidean Algorithm:

$$\begin{aligned} (7) \quad 323 &= 2(124) + 75 \\ (6) \quad 124 &= 1(75) + 49 \\ (5) \quad 75 &= 1(49) + 26 \\ (4) \quad 49 &= 1(26) + 23 \\ (3) \quad 26 &= 1(23) + 3 \\ (2) \quad 23 &= 7(3) + 2 \\ (1) \quad 3 &= 1(2) + 1 \\ (0) \quad 2 &= 2(1) + 0 \end{aligned}$$

so $\gcd(154, 105) = 1$

Working Backwards. . . .

$$\begin{aligned} \text{by (1)} \quad 1 &= 3 - 1(2) \\ \text{by (2)} \quad 2 &= 23 - 7(3) \\ \text{so } 1 &= 3 - 1(23 - 7(3)) \\ &= 8(3) - 23 \\ \text{by (3)} \quad 3 &= 26 - 1(23) \\ \text{so } 1 &= 8(26 - 1(23)) - 23 \\ &= 8(26) - 9(23) \\ \text{by (4)} \quad 23 &= 49 - 1(26) \\ \text{so } 1 &= 8(26) - 9(49 - 26) \\ &= 17(26) - 9(49) \\ \text{by (5)} \quad 26 &= 75 - 1(49) \\ \text{so } 1 &= 17(75 - 49) - 9(49) \\ &= 17(75) - 26(49) \\ \text{by (6)} \quad 49 &= 124 - 1(75) \\ \text{so } 1 &= 17(75) - 26(124 - 75) \\ &= 43(75) - 26(124) \\ \text{by (7)} \quad 75 &= 323 - 2(124) \\ \text{so } 1 &= 43(323 - 2(124)) - 26(124) \\ &= 43(323) - 112(124) \end{aligned}$$

Linear Combination Example IV

Find a linear combination of 2002 and 2339 which equals their gcd.

Find $\gcd(2002, 2339)$:

Working Backwards...

Other Integer Results

- ▶ **Lemma 1.** If a, b , and $c \in \mathbb{Z}^+$ such that $\gcd(a,b) = 1$ and $a \mid bc$, then $a \mid c$.
- ▶ **Lemma 2.** If p is a prime and $p \mid a_1 a_2 \dots a_n$ where each $a_i \in \mathbb{Z}$, then $p \mid a_i$ for some i .
- ▶ **Theorem 2.** Let $m \in \mathbb{Z}^+$ and let a, b , and $c \in \mathbb{Z}$. If $ac \equiv bc \pmod{m}$ and $\gcd(c, m) = 1$, then $a \equiv b \pmod{m}$.

Linear Congruence & Inverse

- ▶ **Linear Congruence:** Let $m \in \mathbb{Z}^+$, a, b , and $c \in \mathbb{Z}$, and x be a variable. Then a linear congruence is a congruence of the form $ax \equiv b \pmod{m}$.
- ▶ **Inverse of a Modulus:** an integer, \bar{a} (if it exists) such that $\bar{a} \equiv 1 \pmod{m}$.
- ▶ **Theorem 3.** 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 \bar{a} less than m that is an inverse of a modulo m and every other inverse of a modulo m is congruent to \bar{a} modulo m .

Modular Inverse

Find an inverse of 3 modulo 7.

Since $\gcd(3,7) = 1$, Theorem 2 indicates that an inverse of 3 modulo 7 exists.

The Euclidean algorithm ends quickly when used to find $\gcd(3,7)$:

$$7 = 2(3) + 1$$

and from this equation, we find:

$$-2(3) + 1(7) = 1$$

This shows that -2 is an inverse of 3 modulo 7.

Note that every integer congruent to -2 modulo 7 is also an inverse of 3 — examples are 5, -9 , and 12.

Solving Congruences

If \bar{a} is an inverse of a modulo m , the congruence $ax \equiv b \pmod{m}$ can be solved by multiplying both sides of the linear congruence by \bar{a} .

Example: what are the solutions of the linear congruence $3x \equiv 4 \pmod{7}$?

Solution: By the last example, we know that -2 is an inverse of 3 modulo 7.

Multiplying both sides of the congruence by -2 gives:

$$-2(3)x \equiv -2(4) \pmod{7}$$

Since $-6 \equiv 1 \pmod{7}$ and $-8 \equiv 6 \pmod{7}$, it follows that if x is a solution, then $x \equiv -8 \equiv 6 \pmod{7}$

Furthermore...

We need to determine whether every x with $x \equiv 6 \pmod{7}$ is a solution.

Assume $x \equiv 6 \pmod{7}$. By a previous theorem, it follows that $3x \equiv 3(6) = 18 \equiv 4 \pmod{7}$

which shows that all such x satisfy the congruence.

Thus the solutions to the congruence are the integers x such that $x \equiv 6 \pmod{7} - 6, 13, 20, \dots$, and $-1, -8, -15, \dots$

The **Chinese Remainder Theorem** states that when the moduli of a system of linear congruences are pairwise relatively prime, there is a unique solution of the system, modulo the product of the moduli.

Theorem 4. (Chinese Remainder Theorem) Let m_1, m_2, \dots, m_n be pairwise relatively prime positive integers. The system of congruences:

$$\begin{aligned}x &\equiv a_1 \pmod{m_1} \\x &\equiv a_2 \pmod{m_2} \\&\vdots \\x &\equiv a_n \pmod{m_n}\end{aligned}$$

has a unique solution modulo $m = m_1 m_2 \dots m_n$.

That is, there is a solution x , with $0 \leq x < m$, and all other solutions are congruent modulo m to this solution.

An Example

Solve the system of congruences:

$$\begin{aligned}x &\equiv 2 \pmod{3} \\x &\equiv 3 \pmod{5} \\x &\equiv 2 \pmod{7}\end{aligned}$$

In other words, what number gives a remainder 2 when divided by 3, a remainder of 3 when divided by 5, and a remainder of 2 when divided by 7.

First, let $m = 3(5)(7) = 105$. Then:

$$\begin{aligned}M_1 &= \frac{m}{3} = 35 \\M_2 &= \frac{m}{5} = 21 \\M_3 &= \frac{m}{7} = 15\end{aligned}$$

We see that 2 is an inverse of $M_1 = 35 \pmod{3}$, since $35 \equiv 2 \pmod{3}$

1 is an inverse of $M_2 = 21 \pmod{5}$, since $21 \equiv 1 \pmod{5}$

1 is an inverse of $M_3 = 15 \pmod{7}$, since $15 \equiv 1 \pmod{7}$

The solutions to this system are those x such that:

$$\begin{aligned}x &\equiv a_1 M_1 y_1 + a_2 M_2 y_2 + a_3 M_3 y_3 \\&= 2(35)(2) + 2(21)(1) + 2(15)(1) \pmod{105} \\&= 233 \equiv 23 \pmod{105}\end{aligned}$$

It follows that 23 is the smallest positive integer that is a simultaneous solution.

We conclude that 23 is the smallest positive integer that leaves a remainder of 2 when divided by 3, a remainder of 3 when divided by 5, and a remainder of 2 when divided by 7.