01204211 Discrete Mathematics
Lecture 8a: Integers and GCD

ห.ร.ม.

LCM

ค.ร.น.

Jittat Fakcharoenphol

October 1, 2024

Number theory: integers and divisibility

In the third part of the course, we study number theory, a once-thought-to-be "useless" branch of mathematics.

Number theory: integers and divisibility

In the third part of the course, we study number theory, a once-thought-to-be "useless" branch of mathematics.

Why?

Number theory: integers and divisibility

In the third part of the course, we study number theory, a once-thought-to-be "useless" branch of mathematics.

Why?

- The topic itself is very very beautiful.
- It has many applications in cryptography and error correcting codes.

Number theory: integers and divisibility

In the third part of the course, we study number theory, a once-thought-to-be "useless" branch of mathematics.

Why?

- The topic itself is very very beautiful.
- It has many applications in cryptography and error correcting codes.

We will cover:

- Basic concepts of divisibility, prime numbers, and congruence.
- How to quickly check if a number is prime. *
- How to essentially perform "division" with integers, allowing us to work with important and useful objects like polynomials using only integers.

Number theory: integers and divisibility

In the third part of the course, we study number theory, a once-thought-to-be "useless" branch of mathematics.

Why?

- The topic itself is very very beautiful.
- It has many applications in cryptography and error correcting codes.

We will cover:

- Basic concepts of divisibility, prime numbers, and congruence.
- How to quickly check if a number is prime.
- How to essentially perform "division" with integers, allowing us to work with important and useful objects like polynomials using only integers.
- Applications like cryptography (RSA), secret sharing, erasure codes and error correcting codes.

Definitions

Definition (divisibility)

We say that an integer a **divides** b or b **is divisible by** a if there exist an integer k such that

$$b = ak.$$

If it is the case, we also write $a|b$. We also say that a is a **divisor** (or a **factor**) of b. On the other hand if a does not divide b, we write $a \nmid b$.

$a|b$

Examples

If $a|b$ and $a|c$, prove that $a|(b+c)$.

Proof:

1. เนื่องจาก $a|b$ จะมี k ที่เป็น จน.เต็ม ที่ $b=ka$
2. เนื่องจาก $a|c$ จะมี ℓ ที่เป็น จน.เต็ม ที่ $c=\ell a$
3. จาก ① และ ② จะได้ว่า $b+c = ka + \ell a = (k+\ell)a$
4. เนื่องจาก k และ ℓ เป็น จนเต็ม $k+\ell$ จึงเป็น จนเต็มด้วย
5. จากนิยาม ของ การหารลงตัว จาก 4 จึงสรุปได้ว่า $a|(b+c)$ $\square$

exercise: If $a|b$ and $b|c$, then $a|c$.

1. เนื่องจาก $a|b$, จะมี จน.เต็ม k_a ที่ $b=k_a \cdot a$
2. เนื่องจาก $b|c$, จะมี จน.เต็ม k_b ที่ $c=k_b \cdot b$
3. จาก 1 & 2, $c=k_b \cdot b = k_b \cdot k_a \cdot a$ และ $k_b \cdot k_a$ เป็น จน.เต็ม
4. ดังนั้น $a|c$

Examples

If $a|b$ and $a|c$, prove that $a|(b+c)$.

If $a|b$ and $b|c$, prove that $a|c$.

Remainder

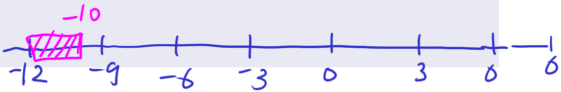
เศษ

Defintion (remainder)

The **remainder** of the division of b with a is an integer r when there exists an integer q such that

$$b = qa + r,$$

where $0 \leq r < |a|$.



เศษของการหาร 10 ด้วย 2 = 1

เศษของการหาร −10 ด้วย 3 = $\boxed{2}$

เศษของการหาร 10 ด้วย −3 = $\boxed{1}$

$$-10 = (4)(-3) + 2$$

$$\boxed{10 = (-3)(-3) + \boxed{1}}$$

Remainder

Defintion (remainder)

The **remainder** of the division of b with a is an integer r when there exists an integer q such that

$$b = qa + r,$$

where $0 \leq r < a$.
We refer to q as the **quotient** of the division.

Remainder

Defintion (remainder)

The **remainder** of the division of b with a is an integer r when there exists an integer q such that

$$b = qa + r,$$

where $0 \leq r < a$.
We refer to q as the **quotient** of the division.

Examples:

Remainder

Defintion (remainder)

The **remainder** of the division of b with a is an integer r when there exists an integer q such that

$$b = qa + r,$$

where $0 \leq r < a$.
We refer to q as the **quotient** of the division.

Examples:

$10 \bmod 3 = 1$

$-10 \bmod 3 = 2$

$10 \bmod (-3) = 1$???

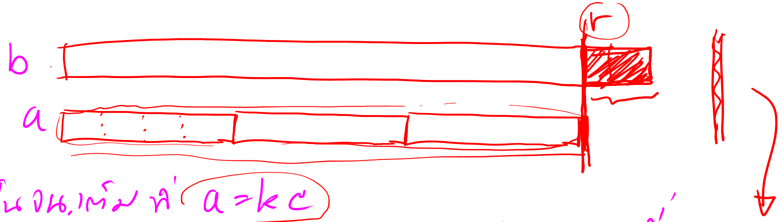
We use operator `mod` to denote an operation for finding the remainder of a division.
I.e., $a \bmod b$ is the remainder of dividing a with b.

Examples

พิจารณา $a > 0$

Let r be the remainder of the division of b by a. Assume that $c|a$ and $c|b$. Prove that $c|r$.

$r = b \bmod a$, If $c|a$ and $c|b$, then $c|r$.



มี k ที่เป็นจำนวนเต็ม ที่ $a = kc$

เนื่องจาก $r = b \bmod a$, มี integer q ที่

$$b = aq + r = kc \cdot q + r$$

เนื่องจาก $c|b$ มีจำนวนเต็ม ℓ ที่ $b = \ell c$ แทนกัน

$$\ell c = k \cdot c \cdot q + r$$

$$r = \ell c - kcq$$

$$= (\ell - kq)c,$$

More examples

For every integer a, $a - 1 | a^2 - 1$.

[illegible] $a^2 - 1 = \underline{(a+1)}(a-1)$

Primes

Definition (primes)

1 ไม่ใช่ จน.เฉพาะ

- An integer $\boxed{p > 1}$ is a **prime** if its divisors are only p, $-p$, 1, and -1.
- If an integer $n > 1$ is not a prime, it is called a **composite**.
- Note: 1 is not a prime and also not a composite.

Fundamental theorem of arithmetic

Unique factorization

Every integer greater than 1 can be represented uniquely as a product of prime numbers, up to the order of the factors.

$$15 = 5 \cdot 3 = 3 \cdot 5$$

Algorithm for testing primes

$2, 3, 4, \ldots\ldots, n-1$

$\boxed{O(n)}$ ×

Recall our `CheckPrime2` algorithm

```
Algorithm CheckPrime2(n):  // Input: an integer n
    if n <= 1:
        return False
    let s = square root of n
    i = 2
    while i <= s:
        if n is divisible by i:
            return False
        i = i + 1
    return True
```

$O(\sqrt{n})$

$= O(2^{\log_2 n / 2})$

How fast can it run?

Input: n

ขนาดของ input (จำนวนบิต)

$\boxed{\log_2 n}$

4

RSA

Algorithm for testing primes

Recall our `CheckPrime2` algorithm

```
Algorithm CheckPrime2(n):  // Input: an integer n
    if n <= 1:
        return False
    let s = square root of n
    i = 2
    while i <= s:
        if n is divisible by i:
            return False
        i = i + 1
    return True
```

How fast can it run? Note that $s = \sqrt{n}$; therefore, it takes time $O(\sqrt{n})$ to run.

Efficient algorithms

Is $O(\sqrt{n})$ for checking a prime number efficient?

Efficient algorithms

Is $O(\sqrt{n})$ for checking a prime number efficient?
What is the "size" of the input to the problem?

Efficient algorithms

Is $O(\sqrt{n})$ for checking a prime number efficient?
What is the "size" of the input to the problem? The input contains one integer n; is the size of the input just 1?

Efficient algorithms

Is $O(\sqrt{n})$ for checking a prime number efficient?
What is the "size" of the input to the problem? The input contains one integer n; is the size of the input just 1?
When working with input consisting only a few numbers, we typically use the number of bits. For integer n, the number of bits of n is $\lceil \log_2 n \rceil$.

Efficient algorithms

Is $O(\sqrt{n})$ for checking a prime number efficient?
What is the "size" of the input to the problem? The input contains one integer n; is the size of the input just 1?
When working with input consisting only a few numbers, we typically use the number of bits. For integer n, the number of bits of n is $\lceil \log_2 n \rceil$.

n	number of bits of n	$\sqrt{n}$
2	1	1.414
4	2	2
16	4	4
1,024	10	32
1,048,576	20	1,024
1,125,899,906,842,624	50	33,554,432
1,267,650,600,228,229,401,496,703,205,376	100	1,125,899,906,842,624

Efficient algorithms

Is $O(\sqrt{n})$ for checking a prime number efficient?
What is the "size" of the input to the problem? The input contains one integer n; is the size of the input just 1?
When working with input consisting only a few numbers, we typically use the number of bits. For integer n, the number of bits of n is $\lceil \log_2 n \rceil$.

n	number of bits of n	$\sqrt{n}$
2	1	1.414
4	2	2
16	4	4
1,024	10	32
1,048,576	20	1,024
1,125,899,906,842,624	50	33,554,432
1,267,650,600,228,229,401,496,703,205,376	100	1,125,899,906,842,624

Side note: Recall that the first step in RSA is to find a pair of large primes. Typically we want them to be of size in the *thousand* bits.

(Fermat's Little Theorem)

Greatest Common Divisors (GCD)

Definition (GCD)

For integers x and y, the **greatest common divisor** (or GCD) of x and y is the largest integer g such that $g|x$ and $g|y$. We refer to it as $gcd(x, y)$.

Greatest Common Divisors (GCD)

Definition (GCD)

For integers x and y, the **greatest common divisor** (or GCD) of x and y is the largest integer g such that $g|x$ and $g|y$. We refer to it as $gcd(x, y)$.

A simple way to find $gcd(x, y)$: $x > 0, \; y > 0$

$gcd(x,y) \geq 1$

```
g = min(x,y)
while (x mod g != 0) or (y mod g != 0):
  g -= 1
return g
```

$O(\min(x,y))$

Greatest Common Divisors (GCD)

Definition (GCD)

For integers x and y, the **greatest common divisor** (or GCD) of x and y is the largest integer g such that $g|x$ and $g|y$. We refer to it as $gcd(x, y)$.

A simple way to find $gcd(x, y)$:

```
g = min(x,y)
while (x mod g != 0) or (y mod g != 0):
  g -= 1
return g
```

What is the running time of this algorithm?

Greatest Common Divisors (GCD)

Definition (GCD)

For integers x and y, the **greatest common divisor** (or GCD) of x and y is the largest integer g such that $g|x$ and $g|y$. We refer to it as $gcd(x, y)$.

A simple way to find $gcd(x, y)$:

```
g = min(x,y)
while (x mod g != 0) or (y mod g != 0):
  g -= 1
return g
```

What is the running time of this algorithm? Does it run in polynomial time on the size of the input?

Euclid's algorithm

```
Algorithm Euclid(x,y):
  if x mod y == 0:
    return y
  else:
    return Euclid(y, x mod y)
```

x y

y $x \bmod y$

⋮

2400, 1250

1250, 1150

1150, 100

100 50

Euclid's algorithm

```
Algorithm Euclid(x,y):
  if x mod y == 0:
    return y
  else:
    return Euclid(y, x mod y)
```

$= \boxed{gcd(x,y)}$

$x<y$, on $E(y, \cancel{x \bmod y}\ x)$

I.H

$= \boxed{gcd(y, x \bmod y)}$

Let's see how it works with $Euclid(12311, 24324)$:

Euclid(12311, 24324)
Euclid(24324, 12311)
Euclid(12311, 12013)
Euclid(12013, 298)
Euclid(298, 93)
Euclid(93, 19)
Euclid(19, 17)
Euclid(17, 2)
Euclid(2, 1)

Proofs

ผลลัพธ์ของ algo

หรม ตามนิยาม

We have to prove two properties:

- For any integers x and y, $\mathrm{Euclid}(x, y) = gcd(x, y)$.
- The running time of Euclid.

Proofs

We have to prove two properties:

- For any integers x and y, $\mathrm{Euclid}(x, y) = gcd(x, y)$.
- The running time of Euclid.

Note that when $x < y$, $\mathrm{Euclid}(x, y)$ just calls itself with both arguments swapped, i.e., $\mathrm{Euclid}(y, x)$. After that, in each call, x is always larger than y. For simplicity of the analysis, we shall work only with the case that $x > y$.

Theorem 1

For any integers x and y such that $x > y$, $\text{Euclid}(x, y) = gcd(x, y)$.

Proof.

We prove using strong induction. For the base case, note that when $y|x$, $gcd(x, y) = y$; therefore, the base case of the algorithm is correct.
Our induction hypothesis is: for any $x' < x$ and $y' < y$, $\text{Euclid}(x', y') = gcd(x', y')$.
Now assume that $y \nmid x$. The Euclid algorithm returns $\text{Euclid}(y, x \bmod y)$ as the gcd. Note that $y < x$ and $x \bmod y < y$. Therefore, we can use the I.H. to claim that

$$\text{Euclid}(y, x \bmod y) = gcd(y, x \bmod y).$$

Thus, we are left to show that

$$gcd(x, y) = gcd(y, x \bmod y).$$

$\square$

What is $x \bmod y$?

What is $x \bmod y$?

Let $\lfloor a \rfloor$ be the largest integer a' such that $a' \leq \lfloor a \rfloor$.

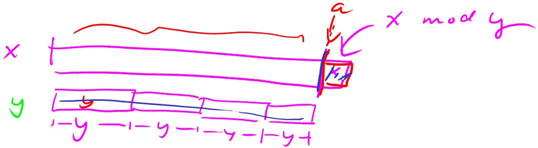
What is $x \bmod y$?

Let $\lfloor a \rfloor$ be the largest integer a' such that $a' \leq \lfloor a \rfloor$.

$$x \bmod y = x - \left\lfloor \frac{x}{y} \right\rfloor \cdot y$$

Lemma 2

If $a|x$ and $a|y$, then $a|x \bmod y$. พิสูจน์ไปแล้ว



Lemma 2

If $a|x$ *and* $a|y$, *then* $a|x \bmod y$.

Lemma 3

$gcd(x, y) = gcd(y, x \bmod y)$

- ให้ g เป็นตัวหารร่วมของ x, y จาก lem 2, g เป็นตัวหารร่วมของ $y, x \bmod y$

$$\Rightarrow gcd(y, x \bmod y) \geq gcd(x, y)$$

- ให้ g เป็นตัวหารร่วมของ $y, x \bmod y$, (ฝากไว้), g จะหาร x ลงตัวด้วย.

g จะเป็นตัวหารร่วมของ x & y

$$\Rightarrow gcd(x, y) \geq gcd(y, x \bmod y)$$

How many recursive calls does Euclid's algorithm makes?

How many recursive calls does Euclid's algorithm makes?

Consider Euclid(x, y):

- If we start with $x < y$, the next calls will always have that $x > y$; so we have at most one call with $x < y$.

How many recursive calls does Euclid's algorithm makes?

Consider Euclid(x, y):

- If we start with $x < y$, the next calls will always have that $x > y$; so we have at most one call with $x < y$.
- When can we decrease the value of x or y in the calls?

How many recursive calls does Euclid's algorithm makes?

Consider Euclid(x, y):

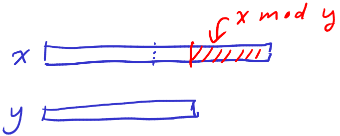
- If we start with $x < y$, the next calls will always have that $x > y$; so we have at most one call with $x < y$.
- When can we decrease the value of x or y in the calls?
- When $y \leq x/2$, when Euclid(x, y) calls Euclid$(y, x \bmod y)$ the first argument decreases by half.

How many recursive calls does Euclid's algorithm makes?

Consider Euclid(x, y):

- If we start with $x < y$, the next calls will always have that $x > y$; so we have at most one call with $x < y$.
- When can we decrease the value of x or y in the calls?
- When $y \leq x/2$, when Euclid(x, y) calls Euclid$(y, x \bmod y)$ the first argument decreases by half.
- How about when $y > x/2$?

$x \bmod y < x/2$



Euclid(x, y)

$\downarrow$

Euclid$(y, x \bmod y)$ ①

$\downarrow$

Euclid$(x \bmod y, y \bmod (x \bmod y))$ ②

How many recursive calls does Euclid's algorithm makes?

Consider Euclid(x, y):

- If we start with $x < y$, the next calls will always have that $x > y$; so we have at most one call with $x < y$.
- When can we decrease the value of x or y in the calls?
- When $y \leq x/2$, when Euclid(x, y) calls Euclid$(y, x \bmod y)$ the first argument decreases by half.
- How about when $y > x/2$?
  Euclid$(x, y) \Rightarrow$

How many recursive calls does Euclid's algorithm makes?

Consider Euclid(x, y):

- If we start with $x < y$, the next calls will always have that $x > y$; so we have at most one call with $x < y$.
- When can we decrease the value of x or y in the calls?
- When $y \leq x/2$, when Euclid(x, y) calls Euclid$(y, x \bmod y)$ the first argument decreases by half.
- How about when $y > x/2$?
  Euclid$(x, y) \Rightarrow$ Euclid$(y, x \bmod y) \Rightarrow$

How many recursive calls does Euclid's algorithm makes?

Consider Euclid(x, y):

- If we start with $x < y$, the next calls will always have that $x > y$; so we have at most one call with $x < y$.
- When can we decrease the value of x or y in the calls?
- When $y \leq x/2$, when Euclid(x, y) calls Euclid$(y, x \bmod y)$ the first argument decreases by half.
- How about when $y > x/2$?
  Euclid$(x, y) \Rightarrow$ Euclid$(y, x \bmod y) \Rightarrow$ Euclid$(x \bmod y, y \bmod (x \bmod y))$

How many recursive calls does Euclid's algorithm makes?

Consider Euclid(x, y):

- If we start with $x < y$, the next calls will always have that $x > y$; so we have at most one call with $x < y$.
- When can we decrease the value of x or y in the calls?
- When $y \leq x/2$, when Euclid(x, y) calls Euclid$(y, x \bmod y)$ the first argument decreases by half.
- How about when $y > x/2$?
  Euclid$(x, y) \Rightarrow$ Euclid$(y, x \bmod y) \Rightarrow$ Euclid$(x \bmod y, y \bmod (x \bmod y))$ Note that in this case, $x \bmod y = x - y \leq x/2$.

How many recursive calls does Euclid's algorithm makes?

Consider Euclid(x, y):

- If we start with $x < y$, the next calls will always have that $x > y$; so we have at most one call with $x < y$.
- When can we decrease the value of x or y in the calls?
- When $y \leq x/2$, when Euclid(x, y) calls Euclid$(y, x \bmod y)$ the first argument decreases by half.
- How about when $y > x/2$?
  Euclid$(x, y) \Rightarrow$ Euclid$(y, x \bmod y) \Rightarrow$ Euclid$(x \bmod y, y \bmod (x \bmod y))$ Note that in this case, $x \bmod y = x - y \leq x/2$. Thus, after two recursive calls, the first argument decreases by half.
- How many times can that happen?
- The first argument can decrease by a factor of two for at most $\log x$ times. Therefore, the Euclid algorithm runs in time $\underline{O(\log \max\{x, y\})} = \underline{O(\log x + \log y)}$.

[illegible] polynomial - time

Computing power (subroutine ของ RSA)

$y = 2^k$

How fast can we compute x^y?

$\underbrace{x \cdot x \cdot x \cdot \cdots \cdot x}_{y}$

$O(y)$ — ค่าคูณ 1 ครั้ง $\boxed{O(1)}$

ไม่เป็น polynomial.

$x^1 = x$

$x \cdot x = x^2$

$x^2 \cdot x^2 = x^4$

$x^4 \cdot x^4 = x^8$

repeated squaring

```
t ← x
while y > 1:
    t ← t * t
    y ← y/2
```

$O(\log y)$

ถ้า y ไม่เป็นกำลังของ 2

$\boxed{14} \to \boxed{7}$

$(x^3)^2$

```
def power(x,y)
    if y == 0:
        return 1
    else
        t = power(x, y/2)
        return t*t
```

Computing power

How fast can we compute x^y?

```
Algorithm power(x,y):
  a = 1
  for i = 1,2,...,y:
    a *= x
  return a
```

Computing power

How fast can we compute x^y?

```
Algorithm power(x,y):
  a = 1
  for i = 1,2,...,y:
    a *= x
  return a
```

What is the running time?

Computing power

How fast can we compute x^y?

```
Algorithm power(x,y):
  a = 1
  for i = 1,2,...,y:
    a *= x
  return a
```

What is the running time? Is it efficient?

Repeated squaring

If y is a power of two, we can find x^y using small number of multiplications using repeated squaring. E.g.,

$$x^{16} = (x^8)^2 = ((x^4)^2)^2 = (((x^2)^2)^2)^2.$$

Repeated squaring

If y is a power of two, we can find x^y using small number of multiplications using repeated squaring. E.g.,

$$x^{16} = (x^8)^2 = ((x^4)^2)^2 = (((x^2)^2)^2)^2.$$

```
Algorithm power(x,y):  // for y=2^k
  if y == 0:
    return 1
  else:
    a = power(x, y / 2)
    return a*a
```

Repeated squaring (general y)

x — b bits, y — b bits — 2^b

```
Algorithm power(x,y):
  if y == 0:
    return 1
  else:
    a = power(x, floor(y / 2))
    if y mod 2 == 0:
      return a*a
    else:
      return a*a*x
```

// y เป็นคู่

ขาดไป x

$y = 2\ell + 1$

$\underbrace{x^\ell}_{a} \cdot \underbrace{x^\ell}_{a} \cdot x$

$\rightarrow \boxed{O(\log y)}$

x — b bits

x^y — $\boxed{b \cdot 2^b}$ bits

$\boxed{x^y \bmod n}$

$(x^{y/2} \bmod n)(x^{y/2} \bmod n) \bmod n$

Repeated squaring (general y)

```
Algorithm power(x,y):
  if y == 0:
    return 1
  else:
    a = power(x, floor(y / 2))
    if y mod 2 == 0:
      return a*a
    else
      return a*a*x
```

What is the number of recursive calls?

Repeated squaring (general y)

```
Algorithm power(x,y):
  if y == 0:
    return 1
  else:
    a = power(x, floor(y / 2))
    if y mod 2 == 0:
      return a*a
    else
      return a*a*x
```

What is the number of recursive calls?
What is the running time?

Repeated squaring (general y)

```
Algorithm power(x,y):
  if y == 0:
    return 1
  else:
    a = power(x, floor(y / 2))
    if y mod 2 == 0:
      return a*a
    else
      return a*a*x
```

What is the number of recursive calls?
What is the running time?
While the number of multiplication is small, the numbers involved is huge as x^y has $y \log x$ bits. Computing x^y exactly definitely takes a long time.

Repeated squaring (general y, mod n) ← colab RSA

$m^e \pmod{n}$

* Computing $x^y \bmod n$:

```
Algorithm power(x,y,n):
  if y == 0:
    return 1
  else:
    a = power(x, floor(y / 2)) mod n
    if y mod 2 == 0:
      return (a*a) mod n
    else
      return (a*a*x) mod n
```

$O(\log y)$ รอบ

$y // 2$

mod %

ตัวเลขจะไม่เกิน $\boxed{n^2}$ → $3 \log n$