Lesson 12. Numerical Algorithms (1): Basic Arithmetic Operations

I111E – Algorithms and Data Structures

Ryuhei Uehara & Giovanni Viglietta
uehara@jaist.ac.jp & johnny@jaist.ac.jp

JAIST – November 27, 2019

All material is available at
www.jaist.ac.jp/~uehara/course/2019/i111e

Topics of today's lecture

- Representation of arbitrarily large integers
- Arithmetic operations with large integers
  - Addition
  - Subtraction
  - Multiplication
  - Integer division (quotient and remainder)
- Modular operations with large integers
  - Modular addition
  - Modular subtraction
  - Modular multiplication
  - Modular exponentiation
  - Euclid's algorithm for greatest common divisor
  - Extended Euclid's algorithm for modular division

Modular arithmetic for cryptography

We will see two applications of modular arithmetic:

- Finding large prime numbers efficiently (next lesson)
- Public-key cryptography, e.g., RSA (in two lessons)

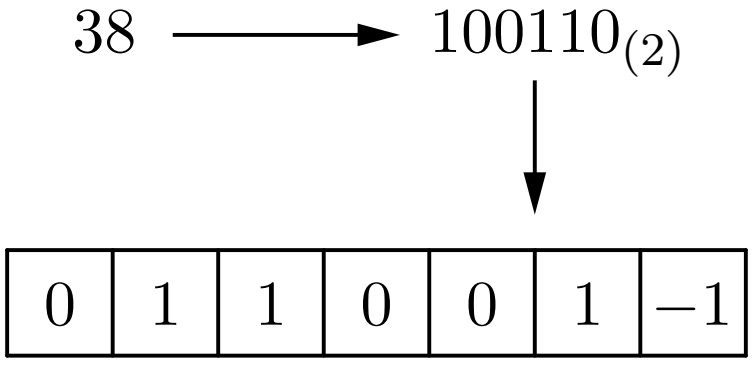
The security of modern cryptosystems relies on these facts:

- Modular exponentiation is easy (today's lesson)
- Computing modular logarithms is hard
- Checking if a number is prime is easy (next lesson)
- Finding the prime factors of a number is hard

Representing large integers

The standard C types have fixed size: for instance, an **`int`** variable can typically store numbers from $-2{,}147{,}483{,}648$ to $2{,}147{,}483{,}647$. We want to work with much larger (unboundedly large) integers!

Idea: represent an integer as an array of (binary) digits, terminating with -1.



Notice that less significant bits come first.

(We will see why this is a good choice...)

Representing large integers

```
char zero[] = {0, -1};
char one[] = {1, -1};
char two[] = {0, 1, -1};
char four[] = {0, 0, 1, -1};
char thousand[] = {0, 0, 0, 1, 0, 1, 1, 1, 1, -1};
```

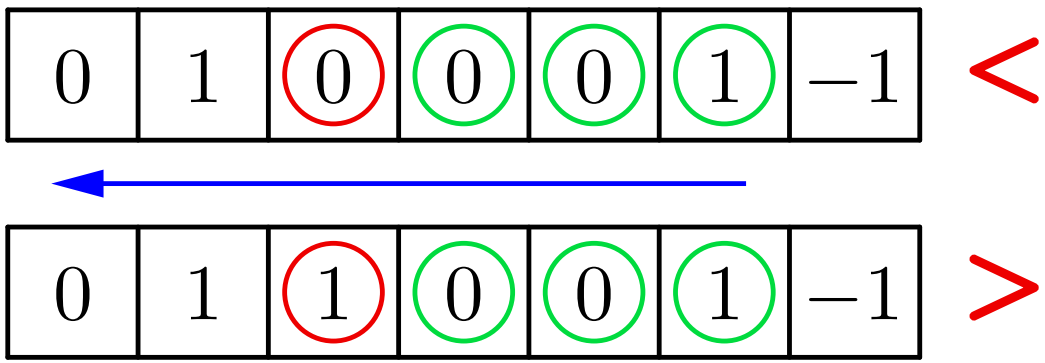
Note that the size of a number is not its value,

but the number of (binary) digits needed to represent it.

```
int num_length(char* a) {
    int i = 0;
    while (a[i] != -1) i++;
    return i;
}
```

Comparing large integers

To compare two numbers, we compare their digits in order, starting from the most significant ones:



Comparing large integers

Our function returns 1 if $a < b$, 0 if $a = b$, and -1 if $a > b$.

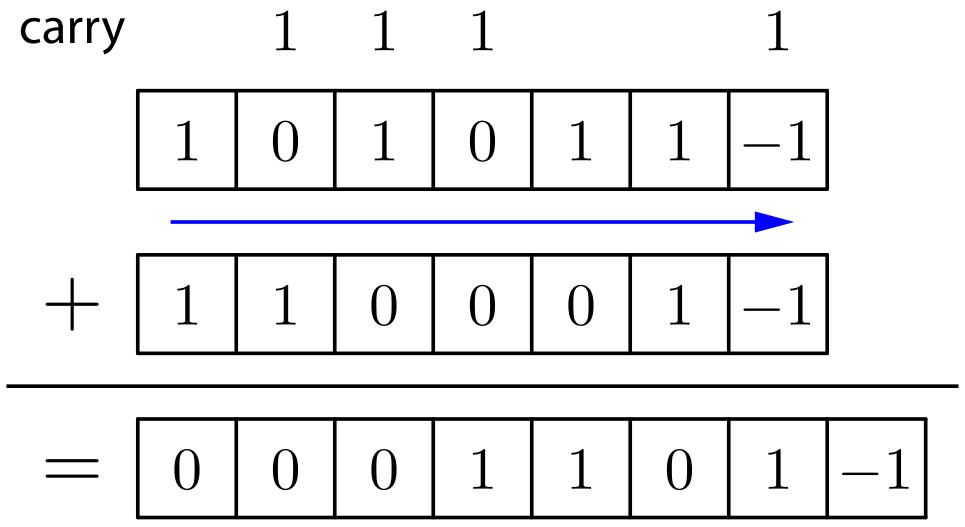
```
int compare(char* a, char* b) {
    int la = num_length(a);
    int lb = num_length(b);
    for (int i = max(la, lb) - 1; i >= 0; i--) {
        int x = 0; if (i < la) x = a[i];
        int y = 0; if (i < lb) y = b[i];
        if (x < y) return 1;
        if (x > y) return -1;
    }
    return 0;
}
```

The algorithm runs in $O(n)$ time.

This is optimal, because to compare two numbers we need at least to read their digits (which already takes linear time).

Addition

To add two numbers, we can use the grade-school algorithm:



If the numbers have size n, the result has size at most $n + 1$.

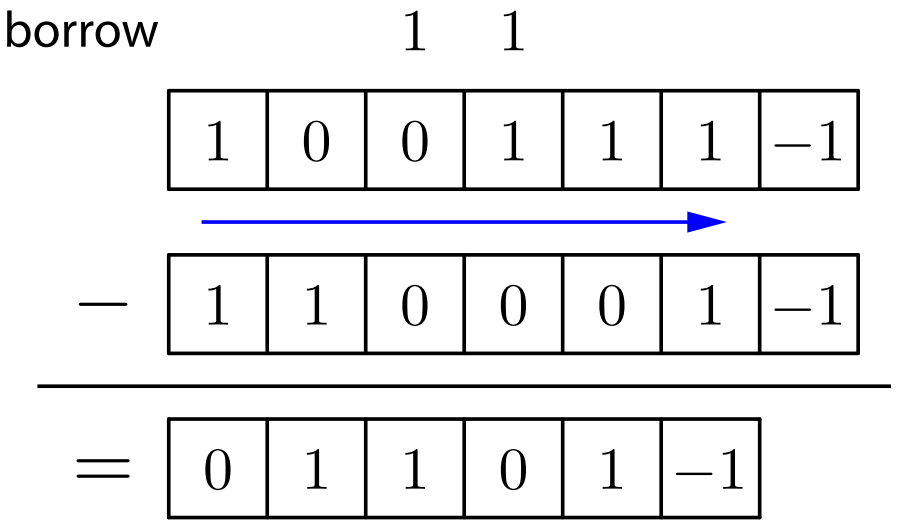
Addition

```
char* add(char* a, char* b, int base) {
    int la = num_length(a);
    int lb = num_length(b);
    char* c = malloc(max(la, lb) + 2);
    int carry = 0;
    int i = 0;
    while (i < la || i < lb || carry != 0) {
        int x = 0; if (i < la) x = a[i];
        int y = 0; if (i < lb) y = b[i];
        c[i] = x + y + carry;
        if (c[i] >= base) {
            c[i] -= base;
            carry = 1;
        }
        else carry = 0;
        i++;
    }
    c[i] = -1;
    return c;
}
```

The algorithm runs in $O(n)$ time (optimal).

Subtraction

Again, we use the grade-school algorithm:



For our purposes, we may assume that the first number is always greater than the second.

So we do not have to deal with negative numbers.

Subtraction

```
char* sub(char* a, char* b) {
    int la = num_length(a);
    int lb = num_length(b);
    char* c = malloc(la + 1);
    int borrow = 0;
    int i = 0;
    while (i < la) {
        int x = a[i];
        int y = 0; if (i < lb) y = b[i];
        c[i] = x - y - borrow;
        if (c[i] < 0) {
            c[i] += 2;
            borrow = 1;
        }
        else borrow = 0;
        i++;
    }
    do {
        c[i] = -1;
        i--;
    } while (i > 0 && c[i] == 0);
    return c;
}
```

The algorithm runs in $O(n)$ time (optimal).

Multiplication

For multiplication, we use an idea by Al-Khwarizmi:

$$a \cdot b = \begin{cases} (a \cdot \lfloor b/2 \rfloor) \cdot 2 & \text{if } b \text{ is even} \\ (a \cdot \lfloor b/2 \rfloor) \cdot 2 + a & \text{if } b \text{ is odd} \end{cases}$$

So we can first compute $a \cdot \lfloor b/2 \rfloor$ recursively,
and then compute $a \cdot b$ by the formula above.

Multiplication

For multiplication, we use an idea by Al-Khwarizmi:

$$a \cdot b = \begin{cases} (a \cdot \lfloor b/2 \rfloor) \cdot 2 & \text{if } b \text{ is even} \\ (a \cdot \lfloor b/2 \rfloor) \cdot 2 + a & \text{if } b \text{ is odd} \end{cases}$$

So we can first compute $a \cdot \lfloor b/2 \rfloor$ recursively,
and then compute $a \cdot b$ by the formula above.

Note: the following operations can be performed in constant time

- $\lfloor b/2 \rfloor$ can be computed as `b+1`, where `b` is an array of bits.
- The parity of b can be checked by evaluating `b[0]`.

This is why we chose to put the least significant digits first!

Multiplication

```
char* mul(char* a, char* b) {
    if (compare(b, zero) == 0) return zero;
    char* c = mul(a, b + 1);
    c = add(c, c, 2);
    if (b[0] == 0) return c;
    else return add(c, a, 2);
}
```

The algorithm runs in $O(n^2)$ time.

This is not optimal, but it is sufficiently fast for our purposes.

Multiplication

```
char* mul(char* a, char* b) {
    if (compare(b, zero) == 0) return zero;
    char* c = mul(a, b + 1);
    c = add(c, c, 2);
    if (b[0] == 0) return c;
    else return add(c, a, 2);
}
```

The algorithm runs in $O(n^2)$ time.

This is not optimal, but it is sufficiently fast for our purposes.

Note: by executing `c = add(c, c, 2)` without calling `free(c)`, we are causing a "memory leak" (the area of memory originally pointed by `c` is never returned to the operating system). Since resolving memory leaks would make our code much more cumbersome, we chose to ignore this issue in our exposition.

Integer division

To divide two integers a and $b \neq 0$ means to find a *quotient* q and a *remainder* r such that $0 \leq r < b$ and $a = b \cdot q + r$.

Once again we can apply recursion if we first divide $\lfloor a/2 \rfloor$ by b:

$$\lfloor a/2 \rfloor = b \cdot q' + r'.$$

Now we can write a as:

$$a = \begin{cases} (b \cdot q' + r') \cdot 2 & = b \cdot (2q') + (2r') & \text{if } a \text{ is even} \\ (b \cdot q' + r') \cdot 2 + 1 & = b \cdot (2q') + (2r' + 1) & \text{if } a \text{ is odd} \end{cases}$$

So we have found our $q = 2q'$ and $r = 2r'$ or $2r' + 1$.

We only need to check if $r \geq b$, in which case we have to decrease r by b and increase q by 1.

Integer division

```
typedef struct {
    char* q;
    char* r;
} pair;

pair* divmod(char* a, char* b) {
    if (compare(a, zero) == 0) {
        pair* p = malloc(sizeof(pair));
        p -> q = zero;
        p -> r = zero;
        return p;
    }
    pair* p = divmod(a + 1, b);
    p -> r = add(p -> r, p -> r, 2);
    p -> q = add(p -> q, p -> q, 2);
    if (a[0] == 1) p -> r = add(p -> r, one, 2);
    if (compare(p -> r, b) != 1) {
        p -> r = sub(p -> r, b);
        p -> q = add(p -> q, one, 2);
    }
    return p;
}
```

The algorithm runs in $O(n^2)$ time.

Base conversion

To print numbers in a legible way, we need to convert them from base 2 to base 10:

```
char* base2to10(char* a) {
    char* d;
    if (a[0] == 0) d = zero;
    else d = one;
    if (a[1] == -1) return d;
    char* r = base2to10(a + 1);
    r = add(r, r, 10);
    r = add(r, d, 10);
    return r;
}
```

Modular arithmetic

A system for dealing with restricted ranges of integers.

"a *modulo* m" $=$ the remainder when a is divided by m.

In modular arithmetic, we identify all the integers
that are the same modulo m:

$$\cdots \quad -4 \quad -3 \quad -2 \quad -1 \quad 0 \quad 1 \quad 2 \quad 3 \quad 4 \quad 5 \quad \cdots \pmod 3$$

E.g., $1 \equiv 4 \pmod 3$, $-3 \equiv 3 \pmod 3$, $2 \not\equiv 4 \pmod 3$

Formally, $a \equiv b \pmod m$ ("a *is congruent to* b *modulo* m")
means that there exists an integer k such that $a - b = km$.

Cancellation rules of modular arithmetic

Sometimes a congruence can be simplified:

- $d \cdot a \equiv d \cdot b \pmod{d \cdot m} \implies a \equiv b \pmod{m}$

  Proof: $da \equiv db \pmod{dm} \implies da - db = k \cdot dm$

  $\implies d(a-b) = dkm \implies a - b = km \implies a \equiv b \pmod{m}$

Cancellation rules of modular arithmetic

Sometimes a congruence can be simplified:

- $d \cdot a \equiv d \cdot b \pmod{d \cdot m} \implies a \equiv b \pmod{m}$

  Proof: $da \equiv db \pmod{dm} \implies da - db = k \cdot dm$

  $\implies d(a-b) = dkm \implies a - b = km \implies a \equiv b \pmod{m}$

- If d and m have no common prime factors, then

  $d \cdot a \equiv d \cdot b \pmod{m} \implies a \equiv b \pmod{m}$

  Proof: $da \equiv db \pmod{m} \implies d(a-b) = km$

  Since d and m are relatively prime, k must be a multiple of d.

  $d(a-b) = k'dm \implies a - b = k'm \implies a \equiv b \pmod{m}$

Cancellation rules of modular arithmetic

Sometimes a congruence can be simplified:

- $d \cdot a \equiv d \cdot b \pmod{d \cdot m} \implies a \equiv b \pmod{m}$

  Proof: $da \equiv db \pmod{dm} \implies da - db = k \cdot dm$

  $\implies d(a-b) = dkm \implies a - b = km \implies a \equiv b \pmod{m}$

- If d and m have no common prime factors, then

  $d \cdot a \equiv d \cdot b \pmod{m} \implies a \equiv b \pmod{m}$

  Proof: $da \equiv db \pmod{m} \implies d(a-b) = km$

  Since d and m are relatively prime, k must be a multiple of d.

  $d(a-b) = k'dm \implies a - b = k'm \implies a \equiv b \pmod{m}$

- The second cancellation rule may not work if d and m are not relatively prime: e.g., $2 \equiv 6 \pmod{4}$, but $1 \not\equiv 3 \pmod{4}$.

Modular addition, subtraction, and multiplication

To get modular addition, subtraction, and multiplication, we slightly modify the non-modular ones to make sure that the result is between 0 and $m - 1$:

```
char* mod(char* a, char* b) {
    pair* p = divmod(a, b);
    return p -> r;
}

char* addM(char* a, char* b, char* m) {
    char* c = add(a, b, 2);
    if (compare(c, m) != 1) c = sub(c, m);
    return c;
}

char* subM(char* a, char* b, char* m) {
    if (compare(a, b) != 1) return sub(a, b);
    else return sub(add(a, m, 2), b);
}

char* mulM(char* a, char* b, char* m) {
    return mod(mul(a, b), m);
}
```

The running times are the same as their non-modular counterparts.

Modular exponentiation

For modular exponentiation, we use the same idea of multiplication:

$$a^b = \begin{cases} \left(a^{\lfloor b/2 \rfloor}\right)^2 & \text{if } b \text{ is even} \\ \left(a^{\lfloor b/2 \rfloor}\right)^2 \cdot a & \text{if } b \text{ is odd} \end{cases}$$

Modular exponentiation

For modular exponentiation, we use the same idea of multiplication:

$$a^b = \begin{cases} \left(a^{\lfloor b/2 \rfloor}\right)^2 & \text{if } b \text{ is even} \\ \left(a^{\lfloor b/2 \rfloor}\right)^2 \cdot a & \text{if } b \text{ is odd} \end{cases}$$

To prevent our intermediate results from growing too much, we use modular multiplication at every step.

```
char* expM(char* a, char* b, char* m) {
    if (compare(b, zero) == 0) return one;
    char* c = expM(a, b + 1, m);
    c = mulM(c, c, m);
    if (b[0] == 0) return c;
    else return mulM(c, a, m);
}
```

The running time is $O(n^3)$.

Euclid's algorithm

What about modular division? It turns out that dividing in modular arithmetic has a lot to do with finding greatest common divisors.

$\gcd(a, b) =$ the largest integer that divides both a and b.

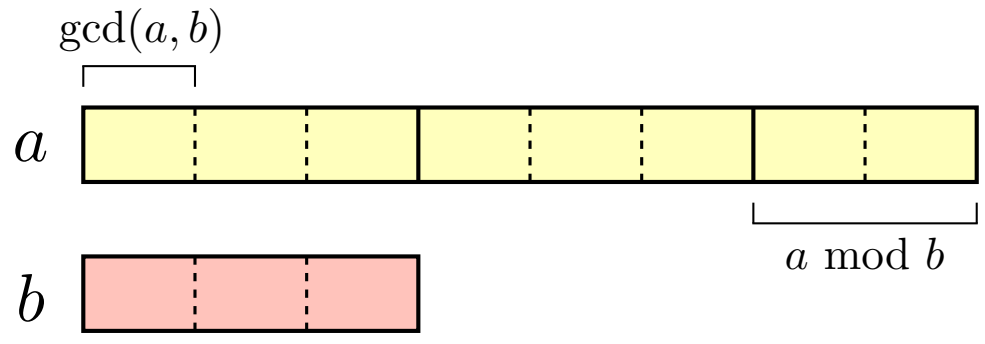
Euclid's algorithm

What about modular division? It turns out that dividing in modular arithmetic has a lot to do with finding greatest common divisors.

$\gcd(a, b) =$ the largest integer that divides both a and b.

Euclid's rule: if $a \geq b$, then $\gcd(a, b) = \gcd(a \bmod b, b)$.

Proof:

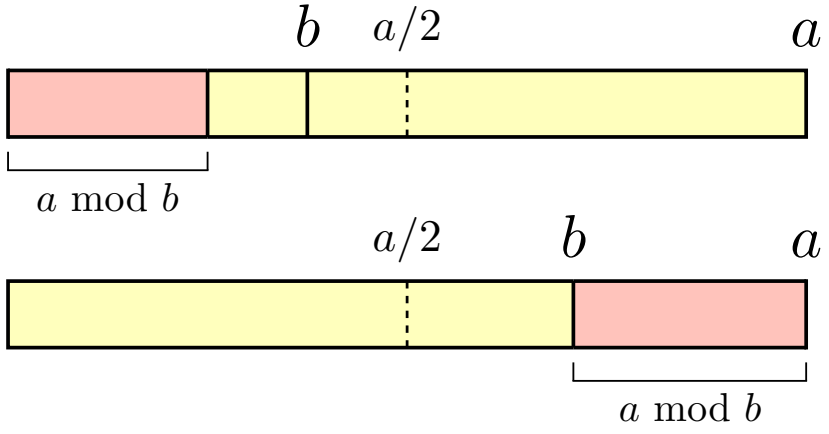


Euclid's algorithm

```
char* euclid(char* a, char* b) {
    if (compare(b, zero) == 0) return a;
    else return euclid(b, mod(a, b));
}
```

To compute the running time, observe that $a \bmod b < a/2$.

Proof: either $b \leq a/2$ (top figure) or $b > a/2$ (bottom figure).



Since the largest argument is halved at each iteration,

it follows that the number of iterations is $O(n)$.

The total running time is therefore $O(n^3)$.

Modular inverse

In real arithmetic, dividing something by a is the same as multiplying it by its *inverse*, $1/a$.

The inverse of a is the number a' such that $a \cdot a' = 1$.

We can extend this definition to the arithmetic modulo m:

the inverse of a is some number a' such that $a \cdot a' \equiv 1 \pmod{m}$.

Modular inverse

In real arithmetic, dividing something by a is the same as multiplying it by its *inverse*, $1/a$.
The inverse of a is the number a' such that $a \cdot a' = 1$.
We can extend this definition to the arithmetic modulo m:
the inverse of a is some number a' such that $a \cdot a' \equiv 1 \pmod{m}$.

- The inverse of a may not exist: 4 modulo 6 has no inverse, because every multiple of 4 is an even number modulo 6.
  If a and m are not relatively prime, a has no inverse mod. m.

Modular inverse

In real arithmetic, dividing something by a is the same as multiplying it by its *inverse*, $1/a$.

The inverse of a is the number a' such that $a \cdot a' = 1$.

We can extend this definition to the arithmetic modulo m:

the inverse of a is some number a' such that $a \cdot a' \equiv 1 \pmod{m}$.

- The inverse of a may not exist: 4 modulo 6 has no inverse, because every multiple of 4 is an even number modulo 6.
  If a and m are not relatively prime, a has no inverse mod. m.
- When it exists, the inverse of a is unique modulo m.
  $\rightarrow$ If a' and a'' are inverses of a, then $aa' \equiv aa'' \pmod{m}$.
  But a and m must be relatively prime, so $a' \equiv a'' \pmod{m}$.

Modular inverse

In real arithmetic, dividing something by a is the same as multiplying it by its *inverse*, $1/a$.
The inverse of a is the number a' such that $a \cdot a' = 1$.
We can extend this definition to the arithmetic modulo m:
the inverse of a is some number a' such that $a \cdot a' \equiv 1 \pmod{m}$.

- The inverse of a may not exist: 4 modulo 6 has no inverse, because every multiple of 4 is an even number modulo 6.
  If a and m are not relatively prime, a has no inverse mod. m.
- When it exists, the inverse of a is unique modulo m.
  $\rightarrow$ If a' and a'' are inverses of a, then $aa' \equiv aa'' \pmod{m}$.
  But a and m must be relatively prime, so $a' \equiv a'' \pmod{m}$.
- What if a and m are relatively prime?
  Does a always have an inverse? If so, how do we compute it?
  $\rightarrow$ The answer is given by Euclid's algorithm!

Extended Euclid's algorithm

Let a be relatively prime with the modulus m, i.e., $\gcd(a, m) = 1$.
We want an a' such that $aa' \equiv 1 \pmod{m}$.
That is, we want two numbers a' and k such that $aa' - km = 1$.
In other words, we want to express the $\gcd$ of a and m
as a *linear combination* of a and m.

$\rightarrow$ We can do it in general, by extending Euclid's algorithm to compute two coefficients x and y such that $ax + by = \gcd(a, b)$.

Extended Euclid's algorithm

Claim: it is possible to extend Euclid's algorithm to compute two coefficients x and y such that $ax + by = \gcd(a, b)$.

Proof: by induction on b. If $b = 0$, we know $\gcd(a, b) = a$, and we may choose $x = 1$ and $y = 0$ as coefficients.

Otherwise, assume the recursive call with arguments b and $a \bmod b$ correctly returned x' and y' such that $bx' + (a \bmod b)y' = \gcd(b, a \bmod b) = \gcd(a, b)$.

By definition of integer division, $a = \lfloor a/b \rfloor \cdot b + (a \bmod b)$.
Substituting $(a - \lfloor a/b \rfloor \cdot b)$ for $(a \bmod b)$, we get
$bx' + (a - \lfloor a/b \rfloor \cdot b)y' = ay' + b(x' - \lfloor a/b \rfloor \cdot y') = \gcd(a, b)$.

So we can set $x = y'$ and $y = x' - \lfloor a/b \rfloor \cdot y'$.

Extended Euclid's algorithm

We can do all computations modulo m,
so we do not have to deal with negative coefficients.

```
typedef struct {
    char* x;
    char* y;
    char* d;
} triplet;

triplet* ext_euclid(char* a, char* b, char* m) {
    if (compare(b, zero) == 0) {
        triplet* t = malloc(sizeof(triplet));
        t -> x = one;
        t -> y = zero;
        t -> d = a;
        return t;
    }
    pair* p = divmod(a, b);
    triplet* t = ext_euclid(b, p -> r, m);
    char* z = mulM(p -> q, t -> y, m);
    z = subM(t -> x, z, m);
    t -> x = t -> y;
    t -> y = z;
    return t;
}
```

The extended Euclid's algorithm still runs in $O(n^3)$ time.

Modular division

We can now use the extended Euclid's algorithm to compute the modular inverse of a number, and use the modular inverse (when it exists) to compute modular division.

```
char* invM(char* a, char* m) {
    triplet* t = ext_euclid(m, a, m);
    if (compare(t -> d, one) == 0) return t -> y;
    else return NULL;
}

char* divM(char* a, char* b, char* m) {
    char* c = invM(b, m);
    if (c == NULL) return NULL;
    else return mulM(a, c, m);
}
```

Both algorithms run in $O(n^3)$ time.

What's the point??

Question:

We started by re-implementing the arithmetic operations from scratch, because we wanted to work with arbitrarily large integers... But now, with modular arithmetic, we are restricting ourselves to a limited range of integers again! What was the point of introducing large integers, then?

What's the point??

Question:

We started by re-implementing the arithmetic operations from scratch, because we wanted to work with arbitrarily large integers... But now, with modular arithmetic, we are restricting ourselves to a limited range of integers again! What was the point of introducing large integers, then?

Answer:

Modern cryptosystems that are based on modular arithmetic use very large moduli (thousands of bits long): the larger the modulus, the safer the system! So, even though the range of integers that we use in cryptography is limited by the modulus, it is still much larger than any standard C variable can handle.