Modular Arithmetic (Part 2)

Ieuan David Vinluan

July 2024

Review of Modular Arithmetic Properties

Let us first review what we know about modular arithmetic and how we can do modular arithmetic in code.

Review of Modular Arithmetic Properties

Let us first review what we know about modular arithmetic and how we can do modular arithmetic in code.

This is the relation for modular addition that we implemented in code.

- $a + b \equiv (a \bmod x + b \bmod x) \pmod{x}$

```
int add(int a, int b, int mod) {
   return (a % mod + b % mod + mod) % mod;
}
```

Review of Modular Arithmetic Properties

Meanwhile, this is the relation we used for modular multiplication.

- $ab \equiv (a \bmod x \cdot b \bmod x) \pmod{x}$

```
int mul(int a, int b, int mod) {
    return (a % mod * b % mod + mod) % mod;
}
```

Modular Exponentiation

With what we know, we can now tackle modular exponentiation.

Recall that exponentiation is just repeated multiplication. Then, from what we know about modular multiplication, we have the following relation for modular exponentiation:

$$\begin{aligned} a^b &\equiv \underbrace{a \cdot a \cdot \ldots \cdot a}_{b \text{ times}} \\ &\equiv \underbrace{(a \bmod x) \cdot (a \bmod x) \cdot \ldots \cdot (a \bmod x)}_{b \text{ times}} \\ &\equiv (a \bmod x)^b \pmod{x} \end{aligned}$$

How can we code this?

Modular Exponentiation

How can we code this?

The obvious solution would be to use a loop, like so:

```
int modpow(int a, int b, int mod) {
    int ret = 1;
    for (int i = 0; i < b; i++) {
        ret = (ret % mod * a % mod + mod) % mod;
    }
    return ret;
}
```

Modular Exponentiation

What is the time complexity of this algorithm?

Modular Exponentiation

What is the time complexity of this algorithm?

The algorithm has a time complexity of $O(b)$ because of the single `for` loop. However, we can still do better!

Modular Exponentiation

An idea is to make use of the fact that $a^b = (a^{b/2})^2$. Instead of computing for a^b directly, we can instead compute $a^{\lfloor b/2 \rfloor}$ and use that computation to get a^b.

Modular Exponentiation

We can implement this idea using recursion!

Modular Exponentiation

We can implement this idea using recursion!

```
int modpow(int a, int b, int mod) {
    if (b == 0) return 1; // base case
    int pre = modpow(a, b / 2, mod);
    if (b % 2 == 0) {
        return (pre % mod * pre % mod + mod) %
            mod;
    } else {
        return (a % mod * pre % mod * pre % mod
             + mod) % mod;
    }
}
```

Modular Exponentiation

Now, what's the time complexity of this algorithm?

Modular Exponentiation

Now, what's the time complexity of this algorithm?

Observe that our recursive calls stop when `b` equals 0, and with each recursive call, we are halving `b`. Thus, if we look at our recursion tree, it will have $\log_2(b)$ levels. Assuming that the integers we are multiplying together are small enough, multiplication takes constant $O(1)$ time.

Modular Exponentiation

Now, what's the time complexity of this algorithm?

Observe that our recursive calls stop when `b` equals 0, and with each recursive call, we are halving `b`. Thus, if we look at our recursion tree, it will have $\log_2(b)$ levels. Assuming that the integers we are multiplying together are small enough, multiplication takes constant $O(1)$ time.

Thus, all in all, our algorithm has a time complexity of $O(\log_2(b))$!

Modular Exponentiation

As intuitive as our recursive function is, we would like a function that uses iteration as much as possible. Remember that recursion is slower due to the overhead of repeated function calls.

Modular Exponentiation

We can translate our recursive function as follows:

```
int modpow(int a, int b, int mod) {
    int ret = 1;
    while (b > 0) {
        if (b % 2 == 1) {
            ret = (ret % mod * a % mod + mod) % mod;
        }
        a = (a % mod * a % mod + mod) % mod;
        b >>= 1;
    }
    return ret;
}
```

Fermat's Little Theorem

Now, we can move onto Fermat's Little Theorem.

Fermat's Little Theorem states that, given an integer a and a prime integer p, the following congruence holds:

$$a^p \equiv a \pmod{p}$$

Fermat's Little Theorem

Now, we can move onto Fermat's Little Theorem.

Fermat's Little Theorem states that, given an integer a and a prime integer p, the following congruence holds:

$$a^p \equiv a \pmod{p}$$

Equivalently, if a and p are coprime:

$$a^{p-1} \equiv 1 \pmod{p}$$

Fermat's Little Theorem

Now, we can move onto Fermat's Little Theorem.

Fermat's Little Theorem states that, given an integer a and a prime integer p, the following congruence holds:

$$a^p \equiv a \pmod{p}$$

Equivalently, if a and p are coprime:

$$a^{p-1} \equiv 1 \pmod{p}$$

For example, if $a = 5$ and $p = 3$, then by Fermat's Little Theorem, $5^3 \equiv 5 \equiv 2 \pmod{3}$, which we can verify.

Fermat's Little Theorem

Example: evaluate $2^{303} \bmod 11$

Fermat's Little Theorem

Example: evaluate $2^{303} \bmod 11$

Solution

By Fermat's Little Theorem, we know that $2^{10} \equiv 1 \pmod{11}$. Then, we can rewrite the congruence as follows:

$$\begin{aligned} 2^{303} &\equiv (2^{10})^{30} \cdot 2^3 \\ &\equiv 1^{30} \cdot 2^3 \\ &\equiv 8 \pmod{11} \end{aligned}$$

Thus, $2^{303} \equiv 8 \pmod{11}$.

Fermat's Little Theorem

Example:
Find the smallest positive integer x such that the following congruence holds: $7x \equiv 1 \pmod{11}$.

Fermat's Little Theorem

Example:
Find the smallest positive integer x such that the following congruence holds: $7x \equiv 1 \pmod{11}$.

Solution

From Fermat's Little Theorem, we know that $7^{10} \equiv 1 \pmod{11}$. From this, we deduce that $7x \equiv 7^{10} \equiv 1 \pmod{11}$, and since 7 and 11 are coprime, $x \equiv 7^9 \pmod{11}$. Since $7^3 \equiv 2 \pmod{11}$, we can simplify the congruence into $x \equiv (7^3)^3 \equiv 2^3 \equiv 8 \pmod{11}$. Thus, $x = 8$.

Modular Inverses

What we did in the last example was find the modular inverse of an integer. For an integer a and a modulus x, its modular inverse is an integer b such that $ab \equiv 1 \pmod{x}$. Essentially, $b \equiv a^{-1} \pmod{x}$.

Modular Inverses

What we did in the last example was find the modular inverse of an integer. For an integer a and a modulus x, its modular inverse is an integer b such that $ab \equiv 1 \pmod{x}$. Essentially, $b \equiv a^{-1} \pmod{x}$.

With the modular inverse, we can do modular division (since division is just multiplying by the multiplicative inverse). Do note, however, that this is only possible if the modular inverse is defined.

Modular Inverses

Given a modulus x, the modular inverse of an integer a is not defined if a and x are not coprime. That is, there exists no integer b such that $ab \equiv 1 \pmod{x}$.

Modular Inverses

Given a modulus x, the modular inverse of an integer a is not defined if a and x are not coprime. That is, there exists no integer b such that $ab \equiv 1 \pmod{x}$.

For example, no integer b satisfies the congruence $2b \equiv 1 \pmod{4}$ because $\gcd(2, 4) = 2$.

Modular Inverses

From our example before, we know that $(a, b, x) = (7, 8, 11)$ satisfies the congruence $ab \equiv 1 \pmod{x}$, with $b = 8$ being the modular inverse of $a = 7$ given a modulus $x = 11$. So, how can we do modular division?

Modular Inverses

From our example before, we know that $(a, b, x) = (7, 8, 11)$ satisfies the congruence $ab \equiv 1 \pmod{x}$, with $b = 8$ being the modular inverse of $a = 7$ given a modulus $x = 11$. So, how can we do modular division?

Let us say we are computing $\frac{343}{7} \bmod 11$. Of course, we know that $\frac{343}{7} \equiv 49 \equiv 5 \pmod{11}$, but using the modular inverse $b = 8$, we can achieve the same result:

$$\begin{aligned}\frac{343}{7} &\equiv 343 \cdot 7^{-1} \\ &\equiv 343 \cdot 8 \\ &\equiv 2 \cdot 8 \\ &\equiv 5 \pmod{11}\end{aligned}$$

Modular Inverses

Now, given some modulus x, how can we compute for the modular inverse b of an integer a?

Modular Inverses

Now, given some modulus x, how can we compute for the modular inverse b of an integer a?

If the modulus x is prime, then finding the modular inverse is simple. From Fermat's Little Theorem, we know that $a^{x-1} \equiv 1 \pmod{x}$ (we can assume a and x are coprime because the modular inverse would not exist otherwise).

Modular Inverses

Now, given some modulus x, how can we compute for the modular inverse b of an integer a?

If the modulus x is prime, then finding the modular inverse is simple. From Fermat's Little Theorem, we know that $a^{x-1} \equiv 1 \pmod{x}$ (we can assume a and x are coprime because the modular inverse would not exist otherwise).

Then, multiplying both sides by a^{-1}, we get that $a^{x-2} \equiv a^{-1} \pmod{x}$, or, equivalently, $a^{-1} \equiv a^{x-2} \pmod{x}$.

Modular Inverses

Thus, translating what we know into code:

Thus, translating what we know into code:

```
// definition of modpow here

int modinv(int a, int p) {
    return modpow(a, p - 2, p);
}
```

Modular Inverses

Now, but what if our modulus x isn't prime?

For this, we will need the Extended Euclidean Algorithm. Given two integer arguments a and b, the algorithm finds two integers x and y such that $ax + by = \gcd(a, b)$, which is known as Bezout's Identity. Since we are trying to find the modular inverse, let us assume that $\gcd(a, b) = 1$.

Modular Inverses

Here is an example of how the Extended Euclidean Algorithm works. Let us find the GCD of 83 and 14 and find the integer coefficients x and y that satisfy $83x + 14y = \gcd(83, 14)$.

Modular Inverses

Here is an example of how the Extended Euclidean Algorithm works. Let us find the GCD of 83 and 14 and find the integer coefficients x and y that satisfy $83x + 14y = \gcd(83, 14)$.

$$(a, b) = (83, 14) \to 83 = 5 \cdot 14 + 13$$

Modular Inverses

Here is an example of how the Extended Euclidean Algorithm works. Let us find the GCD of 83 and 14 and find the integer coefficients x and y that satisfy $83x + 14y = \gcd(83, 14)$.

$$(a, b) = (83, 14) \rightarrow 83 = 5 \cdot 14 + 13$$
$$(a, b) = (14, 13) \rightarrow 14 = 1 \cdot 13 + 1$$

Modular Inverses

Here is an example of how the Extended Euclidean Algorithm works. Let us find the GCD of 83 and 14 and find the integer coefficients x and y that satisfy $83x + 14y = \gcd(83, 14)$.

$$(a, b) = (83, 14) \to 83 = 5 \cdot 14 + 13$$
$$(a, b) = (14, 13) \to 14 = 1 \cdot 13 + 1$$
$$(a, b) = (13, 1) \to 13 = 1 \cdot 13 + 0$$

Modular Inverses

Here is an example of how the Extended Euclidean Algorithm works. Let us find the GCD of 83 and 14 and find the integer coefficients x and y that satisfy $83x + 14y = \gcd(83, 14)$.

$$(a, b) = (83, 14) \to 83 = 5 \cdot 14 + 13$$
$$(a, b) = (14, 13) \to 14 = 1 \cdot 13 + 1$$
$$(a, b) = (13, 1) \to 13 = 1 \cdot 13 + 0$$
$$(a, b) = (1, 0) \to 1 = 0 \cdot 1 + 1$$

Modular Inverses

Here is an example of how the Extended Euclidean Algorithm works. Let us find the GCD of 83 and 14 and find the integer coefficients x and y that satisfy $83x + 14y = \gcd(83, 14)$.

$$
\begin{aligned}
(a, b) &= (83, 14) \to 83 = 5 \cdot 14 + 13 \\
(a, b) &= (14, 13) \to 14 = 1 \cdot 13 + 1 \\
(a, b) &= (13, 1) \to 13 = 1 \cdot 13 + 0 \\
(a, b) &= (1, 0) \to 1 = 0 \cdot 1 + 1
\end{aligned}
$$

Thus, $\gcd(83, 14) = 1$. From here, we can rewrite 1 in terms of 83 and 14, like so:

Modular Inverses

Here is an example of how the Extended Euclidean Algorithm works. Let us find the GCD of 83 and 14 and find the integer coefficients x and y that satisfy $83x + 14y = \gcd(83, 14)$.

$$(a, b) = (83, 14) \to 83 = 5 \cdot 14 + 13$$
$$(a, b) = (14, 13) \to 14 = 1 \cdot 13 + 1$$
$$(a, b) = (13, 1) \to 13 = 1 \cdot 13 + 0$$
$$(a, b) = (1, 0) \to 1 = 0 \cdot 1 + 1$$

Thus, $\gcd(83, 14) = 1$. From here, we can rewrite 1 in terms of 83 and 14, like so:

$$1 = 14 - 1 \cdot 13$$

Modular Inverses

Here is an example of how the Extended Euclidean Algorithm works. Let us find the GCD of 83 and 14 and find the integer coefficients x and y that satisfy $83x + 14y = \gcd(83, 14)$.

$$\begin{aligned}
(a, b) &= (83, 14) \to 83 = 5 \cdot 14 + 13 \\
(a, b) &= (14, 13) \to 14 = 1 \cdot 13 + 1 \\
(a, b) &= (13, 1) \to 13 = 1 \cdot 13 + 0 \\
(a, b) &= (1, 0) \to 1 = 0 \cdot 1 + 1
\end{aligned}$$

Thus, $\gcd(83, 14) = 1$. From here, we can rewrite 1 in terms of 83 and 14, like so:

$$\begin{aligned}
1 &= 14 - 1 \cdot 13 \\
&= 14 - 1 \cdot (83 - 5 \cdot 14)
\end{aligned}$$

Modular Inverses

Here is an example of how the Extended Euclidean Algorithm works. Let us find the GCD of 83 and 14 and find the integer coefficients x and y that satisfy $83x + 14y = \gcd(83, 14)$.

$$
\begin{aligned}
(a, b) &= (83, 14) \to 83 = 5 \cdot 14 + 13 \\
(a, b) &= (14, 13) \to 14 = 1 \cdot 13 + 1 \\
(a, b) &= (13, 1) \to 13 = 1 \cdot 13 + 0 \\
(a, b) &= (1, 0) \to 1 = 0 \cdot 1 + 1
\end{aligned}
$$

Thus, $\gcd(83, 14) = 1$. From here, we can rewrite 1 in terms of 83 and 14, like so:

$$
\begin{aligned}
1 &= 14 - 1 \cdot 13 \\
&= 14 - 1 \cdot (83 - 5 \cdot 14) \\
&= -1 \cdot 83 + 6 \cdot 14
\end{aligned}
$$

Modular Inverses

Here is an example of how the Extended Euclidean Algorithm works. Let us find the GCD of 83 and 14 and find the integer coefficients x and y that satisfy $83x + 14y = \gcd(83, 14)$.

$$
\begin{aligned}
(a, b) &= (83, 14) \to 83 = 5 \cdot 14 + 13 \\
(a, b) &= (14, 13) \to 14 = 1 \cdot 13 + 1 \\
(a, b) &= (13, 1) \to 13 = 1 \cdot 13 + 0 \\
(a, b) &= (1, 0) \to 1 = 0 \cdot 1 + 1
\end{aligned}
$$

Thus, $\gcd(83, 14) = 1$. From here, we can rewrite 1 in terms of 83 and 14, like so:

$$
\begin{aligned}
1 &= 14 - 1 \cdot 13 \\
&= 14 - 1 \cdot (83 - 5 \cdot 14) \\
&= -1 \cdot 83 + 6 \cdot 14
\end{aligned}
$$

Thus, $(x, y) = (-1, 6)$.

Here are some important points to note:

Modular Inverses

Here are some important points to note:

- If we know the coefficients x_n and y_n that satisfy Bezout's Identity for some arguments a_n and b_n, we can do some algebra to find the coefficients x and y for our original arguments a and b

Modular Inverses

Here are some important points to note:

- If we know the coefficients x_n and y_n that satisfy Bezout's Identity for some arguments a_n and b_n, we can do some algebra to find the coefficients x and y for our original arguments a and b
- The final two arguments of the algorithm will always be $a_n = \gcd(a, b) = 1$ and $b_n = 0$, to which the corresponding solution is $(x_n, y_n) = (1, 0)$.

Let's say we know the coefficients x_1 and y_1 that will satisfy Bezout's Identity for some arguments b and $(a \bmod b)$. How can we then find the coefficients x and y that will do the same for a and b?

Modular Inverses

Let's say we know the coefficients x_1 and y_1 that will satisfy Bezout's Identity for some arguments b and $(a \bmod b)$. How can we then find the coefficients x and y that will do the same for a and b?

$$1 = bx_1 + (a \bmod b)y_1$$

$$1 = bx_1 + \left(a - \left\lfloor \frac{a}{b} \right\rfloor \cdot b\right) y_1$$

$$1 = ay_1 + b\left(x_1 - \left\lfloor \frac{a}{b} \right\rfloor \cdot y_1\right)$$

Modular Inverses

Let's say we know the coefficients x_1 and y_1 that will satisfy Bezout's Identity for some arguments b and $(a \bmod b)$. How can we then find the coefficients x and y that will do the same for a and b?

$$1 = bx_1 + (a \bmod b)y_1$$
$$1 = bx_1 + \left(a - \left\lfloor \frac{a}{b} \right\rfloor \cdot b\right) y_1$$
$$1 = ay_1 + b\left(x_1 - \left\lfloor \frac{a}{b} \right\rfloor \cdot y_1\right)$$

Thus, matching the coefficients yields: $x = y_1$ and $y = x_1 - \left\lfloor \frac{a}{b} \right\rfloor \cdot y_1$.

Modular Inverses

Finally, we can translate what we know into code. We can implement the Extended Euclidean Algorithm through recursion!

Modular Inverses

Finally, we can translate what we know into code. We can implement the Extended Euclidean Algorithm through recursion!

```
// returns {x_n, y_n}
pair<int, int> gcd(int a, int b) {
    if (b == 0) {
        return {1, 0};
    }
    auto [x, y] = gcd(b, a % b);
    return {y, x - (a / b) * y};
}

// x is the modulus
int modinv(int a, int x) {
    return gcd(a, x).first; // why?
}
```

Factorials

The factorial of an integer n, denoted by $n!$, is defined as the product of all positive integers less than or equal to n. That is:

$$n! = n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 2 \cdot 1$$

Factorials

The factorial of an integer n, denoted by $n!$, is defined as the product of all positive integers less than or equal to n. That is:

$$n! = n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 2 \cdot 1$$

For example, $3! = 3 \cdot 2 \cdot 1 = 6$, and $5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 = 120$. Note that $0! = 1$.

The factorial function is important in many areas of mathematics and is especially prominent in combinatorics.

Factorials

Many CompProg problems involving combinatorics will require us to precompute a list of factorials of integers from 1 to some integer N, modulo some prime integer p.

Factorials

Many CompProg problems involving combinatorics will require us to precompute a list of factorials of integers from 1 to some integer N, modulo some prime integer p.

From our definition of the factorial, we can note that $n! = n \cdot (n-1)!$, which will be very useful for doing our precomputation.

Factorials

Many CompProg problems involving combinatorics will require us to precompute a list of factorials of integers from 1 to some integer N, modulo some prime integer p.

From our definition of the factorial, we can note that $n! = n \cdot (n-1)!$, which will be very useful for doing our precomputation.

```
vector<int> f(N + 1); // list of factorials
f[0] = 1;
for (int i = 1; i <= N; i++) {
    f[i] = (f[i - 1] % p * i % p + p) % p;
}
```

Factorials

Now that we have talked about factorials, we can now discuss basic permutations and combinations involving distinct items.

Factorials

Now that we have talked about factorials, we can now discuss basic permutations and combinations involving distinct items.

A permutation of a set refers to a specific arrangement of its elements. Here, ordering matters.

Factorials

Now that we have talked about factorials, we can now discuss basic permutations and combinations involving distinct items.

A permutation of a set refers to a specific arrangement of its elements. Here, ordering matters.

For example, the permutations of $\{1, 2, 4\}$ are:

$$1, 2, 4$$
$$1, 4, 2$$
$$2, 1, 4$$
$$2, 4, 1$$
$$4, 1, 2$$
$$4, 2, 1$$

Factorials

The total number of permutations of k elements taken from a set containing n elements is given by ${}^{n}P_{k} = \frac{n!}{(n-k)!}$.

Factorials

The total number of permutations of k elements taken from a set containing n elements is given by ${}^{n}P_k = \frac{n!}{(n-k)!}$.

With our precomputed list of factorials, how can we compute this value, modulo some prime integer p ($p > n$)?

Factorials

Let's assume that we already have all of the factorials until $n!$ precomputed. Then, finding ${}^{n}P_{k}$ is simple:

Factorials

Let's assume that we already have all of the factorials until $n!$ precomputed. Then, finding ${}^{n}P_{k}$ is simple:

1. Retrieve $n! \bmod p$ from our array

Factorials

Let's assume that we already have all of the factorials until $n!$ precomputed. Then, finding ${}^{n}P_{k}$ is simple:

1. Retrieve $n! \bmod p$ from our array
2. Multiply it by the modular inverse of $(n-k)! \bmod p$

Factorials

We can implement this as follows:

Factorials

We can implement this as follows:

```
// put modular exponentiation definition here

// I suggest using a long long instead of an
    int
int permute(int n, int k) {
    int res = f[n];
    res = (res % p * modpow(f[n - k], p - 2, p)
         % p + p) % p;
    return res;
}
```

Factorials

Meanwhile, the combinations of a set refer to a selection of distinct elements from the set. Here, ordering does NOT matter.

Factorials

Meanwhile, the combinations of a set refer to a selection of distinct elements from the set. Here, ordering does NOT matter.

For example, if we are picking 3 elements from the set $\{1, 2, 4, 8\}$, the combinations are:

$$1, 2, 4$$
$$1, 2, 8$$
$$1, 4, 8$$
$$2, 4, 8$$

Factorials

The total number of combinations of k elements taken from a set containing n elements is given by ${}^{n}C_k = \dfrac{n!}{k!(n-k)!}$.

Factorials

The total number of combinations of k elements taken from a set containing n elements is given by ${}^{n}C_k = \dfrac{n!}{k!(n-k)!}$.

Once again, we can compute this value modulo some prime integer p using the same methods that we used to compute permutations. This time, though, we have an additional multiplication of the modular inverse of $k!$.

Factorials

This can be implemented as such:

Factorials

This can be implemented as such:

```
// put modular exponentiation definition here

// Again, I suggest using a long long instead
    of an int
int comb(int n, int k) {
    int res = f[n];
    res = (res % p * modpow(f[n - k], p - 2, p)
         % p + p) % p;
    res = (res % p * modpow(f[k], p - 2, p) % p
         + p) % p;
    return res;
}
```

Additional Resources

Here are some extra links you might find helpful :3

- Binary Exponentiation - Algorithms for Competitive Programming
- Fermat's Little Theorem - Brilliant
- Extended Euclidean Algorithm Explained - Mike the Coder
- Modular multiplicative inverse - GeeksForGeeks
- Combinations and Permutations - Math Is Fun

Homework :3

Check the Reboot website for the homework problems :D