Last update: June 17, 2024 Original

Operations on polynomials and series

Problems in competitive programming, especially the ones involving enumeration some kind, are often solved by reducing the problem to computing something on polynomials and formal power series.

This includes concepts such as polynomial multiplication, interpolation, and more complicated ones, such as polynomial logarithms and exponents. In this article, a brief overview of such operations and common approaches to them is presented.

Basic Notion and facts

In this section, we focus more on the definitions and "intuitive" properties of various polynomial operations. The technical details of their implementation and complexities will be covered in later sections.

Polynomial multiplication

Definition

Univariate polynomial is an expression of form $A(x) = a_0 + a_1 x + \dots + a_n x^n$.

The values $a_0, \dots, a_n$ are polynomial coefficients, typically taken from some set of numbers or number-like structures. In this article, we assume that the coefficients are taken from some field, meaning that operations of addition, subtraction, multiplication and division are well-defined for them (except for division by 0) and they generally behave in a similar way to real numbers.

Typical example of such field is the field of remainders modulo prime number p.

For simplicity we will drop the term *univariate*, as this is the only kind of polynomials we consider in this article. We will also write A instead of $A(x)$ wherever possible, which will be understandable from the context. It is assumed that either $a_n \neq 0$ or $A(x) = 0$.

Definition

The **product** of two polynomials is defined by expanding it as an arithmetic expression:

$$A(x)B(x) = \left(\sum\limits_{i=0}^n a_i x^i \right)\left(\sum\limits_{j=0}^m b_j x^j\right) = \sum\limits_{i,j} a_i b_j x^{i+j} = \sum\limits_{k=0}^{n+m} c_k x^k = C(x).$$

The sequence $c_0, c_1, \dots, c_{n+m}$ of the coefficients of $C(x)$ is called the **convolution** of $a_0, \dots, a_n$ and $b_0, \dots, b_m$.

Definition

The **degree** of a polynomial A with $a_n \neq 0$ is defined as $\deg A = n$.

For consistency, degree of $A(x) = 0$ is defined as $\deg A = -\infty$.

In this notion, $\deg AB = \deg A + \deg B$ for any polynomials A and B.

Convolutions are the basis of solving many enumerative problems.

Example

You have n objects of the first kind and m objects of the second kind.

Objects of first kind are valued $a_1, \dots, a_n$, and objects of the second kind are valued $b_1, \dots, b_m$.

You pick a single object of the first kind and a single object of the second kind. How many ways are there to get the total value k?

Solution

Consider the product $(x^{a_1} + \dots + x^{a_n})(x^{b_1} + \dots + x^{b_m})$. If you expand it, each monomial will correspond to the pair (a_i, b_j) and contribute to the coefficient near $x^{a_i + b_j}$. In other words, the answer is the coefficient near x^k in the product.

Example

You throw a 6-sided die n times and sum up the results from all throws. What is the probability of getting sum of k?

Solution

The answer is the number of outcomes having the sum k, divided by the total number of outcomes, which is 6^n.

What is the number of outcomes having the sum k? For $n = 1$, it may be represented by a polynomial $A(x) = x^1 + x^2 + \dots + x^6$.

For $n = 2$, using the same approach as in the example above, we conclude that it is represented by the polynomial $(x^1 + x^2 + \dots + x^6)^2$.

That being said, the answer to the problem is the k-th coefficient of $(x^1 + x^2 + \dots + x^6)^n$, divided by 6^n.

The coefficient near x^k in the polynomial $A(x)$ is denoted shortly as $[x^k]A$.

Formal power series

Definition

A **formal power series** is an infinite sum $A(x) = a_0 + a_1 x + a_2 x^2 + \dots$, considered regardless of its convergence properties.

In other words, when we consider e.g. a sum $1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \dots = 2$, we imply that it *converges* to 2 when the number of summands approach infinity. However, formal series are only considered in terms of sequences that make them.

Definition

The **product** of formal power series $A(x)$ and $B(x)$, is also defined by expanding it as an arithmetic expression:

$$A(x)B(x) = \left(\sum_{i=0}^{\infty} a_i x^i\right)\left(\sum_{j=0}^{\infty} b_j x^j\right) = \sum_{i,j} a_i b_j x^{i+j} = \sum_{k=0}^{\infty} c_k x^k = C(x),$$

where the coefficients $c_0, c_1, \dots$ are define as finite sums

$$c_k = \sum_{i=0}^{k} a_i b_{k-i}.$$

The sequence $c_0, c_1, \dots$ is also called a **convolution** of $a_0, a_1, \dots$ and $b_0, b_1, \dots$, generalizing the concept to infinite sequences.

Thus, polynomials may be considered formal power series, but with finite number of coefficients.

Formal power series play a crucial role in enumerative combinatorics, where they're studied as generating functions for various sequences. Detailed explanation of generating functions and the intuition behind them will, unfortunately, be out of scope for this article, therefore the curious reader is referenced e.g. here for details about their combinatorial meaning.

However, we will very briefly mention that if $A(x)$ and $B(x)$ are generating functions for sequences that enumerate some objects by number of "atoms" in them (e.g. trees by the number of vertices), then the product $A(x)B(x)$ enumerates objects that can be described as pairs of objects of kinds A and B, enumerates by the total number of "atoms" in the pair.

Example

Let $A(x) = \sum\limits_{i=0}^{\infty} 2^i x^i$ enumerate packs of stones, each stone colored in one of 2 colors (so, there are 2^i such packs of size i) and $B(x) = \sum\limits_{j=0}^{\infty} 3^j x^j$ enumerate packs of stones, each stone colored in one of 3 colors. Then $C(x) = A(x)B(x) = \sum\limits_{k=0}^{\infty} c_k x^k$ would enumerate objects that may be described as "two packs of stones, first pack only of stones of type A, second pack only of stones of type B, with total number of stones being k" for c_k.

In a similar way, there is an intuitive meaning to some other functions over formal power series.

Long polynomial division

Similar to integers, it is possible to define long division on polynomials.

Definition

For any polynomials A and $B \neq 0$, one may represent A as

$$A = D \cdot B + R, \quad \deg R < \deg B,$$

where R is called the **remainder** of A modulo B and D is called the **quotient**.

Denoting $\deg A = n$ and $\deg B = m$, naive way to do it is to use long division, during which you multiply B by the monomial $\frac{a_n}{b_m} x^{n-m}$ and subtract it from A, until the degree of A is smaller than that of B. What remains of A in the end will be the remainder (hence the name), and the polynomials with which you multiplied B in the process, summed together, form the quotient.

Definition

If A and B have the same remainder modulo C, they're said to be **equivalent** modulo C, which is denoted as

$$A \equiv B \pmod{C}.$$

Polynomial long division is useful because of its many important properties:

- A is a multiple of B if and only if $A \equiv 0 \pmod{B}$.
- It implies that $A \equiv B \pmod{C}$ if and only if $A - B$ is a multiple of C.
- In particular, $A \equiv B \pmod{C \cdot D}$ implies $A \equiv B \pmod{C}$.
- For any linear polynomial $x - r$ it holds that $A(x) \equiv A(r) \pmod{x - r}$.
- It implies that A is a multiple of $x - r$ if and only if $A(r) = 0$.
- For modulo being x^k, it holds that $A \equiv a_0 + a_1 x + \dots + a_{k-1} x^{k-1} \pmod{x^k}$.

Note that long division can't be properly defined for formal power series. Instead, for any $A(x)$ such that $a_0 \neq 0$, it is possible to define an inverse formal power series $A^{-1}(x)$, such that $A(x)A^{-1}(x) = 1$. This fact, in turn, can be used to compute the result of long division for polynomials.

Basic implementation

Here you can find the basic implementation of polynomial algebra.

It supports all trivial operations and some other useful methods. The main class is `poly<T>` for polynomials with coefficients of type `T`.

All arithmetic operation `+`, `-`, `*`, `%` and `/` are supported, `%` and `/` standing for remainder and quotient in Euclidean division.

There is also the class `modular<m>` for performing arithmetic operations on remainders modulo a prime number `m`.

Other useful functions:

- `deriv()`: computes the derivative $P'(x)$ of $P(x)$.
- `integr()`: computes the indefinite integral $Q(x) = \int P(x)$ of $P(x)$ such that $Q(0) = 0$.
- `inv(size_t n)`: calculate the first n coefficients of $P^{-1}(x)$ in $O(n \log n)$.
- `log(size_t n)`: calculate the first n coefficients of $\ln P(x)$ in $O(n \log n)$.
- `exp(size_t n)`: calculate the first n coefficients of $\exp P(x)$ in $O(n \log n)$.
- `pow(size_t k, size_t n)`: calculate the first n coefficients for $P^k(x)$ in $O(n \log nk)$.
- `deg()`: returns the degree of $P(x)$.
- `lead()`: returns the coefficient of $x^{\deg P(x)}$.
- `resultant(poly<T> a, poly<T> b)`: computes the resultant of a and b in $O(|a| \cdot |b|)$.
- `bpow(T x, size_t n)`: computes x^n.
- `bpow(T x, size_t n, T m)`: computes $x^n \pmod m$.
- `chirpz(T z, size_t n)`: computes $P(1), P(z), P(z^2), \dots, P(z^{n-1})$ in $O(n \log n)$.
- `vector<T> eval(vector<T> x)`: evaluates $P(x_1), \dots, P(x_n)$ in $O(n \log^2 n)$.
- `poly<T> inter(vector<T> x, vector<T> y)`: interpolates a polynomial by a set of pairs $P(x_i) = y_i$ in $O(n \log^2 n)$.
- And some more, feel free to explore the code!

Arithmetic

Multiplication

The very core operation is the multiplication of two polynomials. That is, given the polynomials A and B:

$$A = a_0 + a_1 x + \dots + a_n x^n$$

$$B = b_0 + b_1 x + \dots + b_m x^m$$

You have to compute polynomial $C = A \cdot B$, which is defined as

$$\boxed{C = \sum\limits_{i=0}^n \sum\limits_{j=0}^m a_i b_j x^{i+j}} = c_0 + c_1 x + \dots + c_{n+m} x^{n+m}.$$

It can be computed in $O(n \log n)$ via the Fast Fourier transform and almost all methods here will use it as subroutine.

Inverse series

If $A(0) \neq 0$ there always exists an infinite formal power series $A^{-1}(x) = q_0+q_1 x + q_2 x^2 + \dots$ such that $A^{-1} A = 1$. It often proves useful to compute first k coefficients of A^{-1} (that is, to compute it modulo x^k). There are two major ways to calculate it.

Divide and conquer

This algorithm was mentioned in Schönhage's article and is inspired by Graeffe's method. It is known that for $B(x)=A(x)A(-x)$ it holds that $B(x)=B(-x)$, that is, $B(x)$ is an even polynomial. It means that it only has non-zero coefficients with even numbers and can be represented as $B(x)=T(x^2)$. Thus, we can do the following transition:

$$A^{-1}(x) \equiv \frac{1}{A(x)} \equiv \frac{A(-x)}{A(x)A(-x)} \equiv \frac{A(-x)}{T(x^2)} \pmod{x^k}$$

Note that $T(x)$ can be computed with a single multiplication, after which we're only interested in the first half of coefficients of its inverse series. This effectively reduces the initial problem of computing $A^{-1} \pmod{x^k}$ to computing $T^{-1} \pmod{x^{\lfloor k/2 \rfloor}}$.

The complexity of this method can be estimated as

$$T(n) = T(n/2) + O(n \log n) = O(n \log n).$$

Sieveking–Kung algorithm

The generic process described here is known as Hensel lifting, as it follows from Hensel's lemma. We'll cover it in more detail further below, but for now let's focus on ad hoc solution. "Lifting" part here means that we start with the approximation $B_0 = q_0 = a_0^{-1}$, which is $A^{-1} \pmod x$ and then iteratively lift from $\bmod x^a$ to $\bmod x^{2a}$.

Let $B_k \equiv A^{-1} \pmod{x^a}$. The next approximation needs to follow the equation $AB_{k+1} \equiv 1 \pmod{x^{2a}}$ and may be represented as $B_{k+1} = B_k + x^a C$. From this follows the equation

$$A(B_k + x^a C) \equiv 1 \pmod{x^{2a}}.$$

Let $AB_k \equiv 1 + x^a D \pmod{x^{2a}}$, then the equation above implies

$$x^a(D + AC) \equiv 0 \pmod{x^{2a}} \implies D \equiv -AC \pmod{x^a} \implies C \equiv -B_k D \pmod{x^a}.$$

From this, one can obtain the final formula, which is

$$x^a C \equiv -B_k x^a D \equiv B_k(1 - AB_k) \pmod{x^{2a}} \implies \boxed{B_{k+1} \equiv B_k(2 - AB_k) \pmod{x^{2a}}}$$

Thus starting with $B_0 \equiv a_0^{-1} \pmod x$ we will compute the sequence B_k such that $AB_k \equiv 1 \pmod{x^{2^k}}$ with the complexity

$$T(n) = T(n/2) + O(n \log n) = O(n \log n).$$

The algorithm here might seem a bit more complicated than the first one, but it has a very solid and practical reasoning behind it, as well as a great generalization potential if looked from a different perspective, which would be explained further below.

Euclidean division

Consider two polynomials $A(x)$ and $B(x)$ of degrees n and m. As it was said earlier you can rewrite $A(x)$ as

$$A(x) = B(x)D(x) + R(x), \deg R < \deg B.$$

Let $n \geq m$, it would imply that $\deg D = n - m$ and the leading $n - m + 1$ coefficients of A don't influence R. It means that you can recover $D(x)$ from the largest $n - m + 1$ coefficients of $A(x)$ and $B(x)$ if you consider it as a system of equations.

The system of linear equations we're talking about can be written in the following form:

$$\begin{bmatrix} a_n \\ \vdots \\ a_{m+1} \\ a_m \end{bmatrix} = \begin{bmatrix} b_m & \dots & 0 & 0 \\ \vdots & \ddots & \vdots & \vdots \\ \dots & \dots & b_m & 0 \\ \dots & \dots & b_{m-1} & b_m \end{bmatrix} \begin{bmatrix} d_{n-m} \\ \vdots \\ d_1 \\ d_0 \end{bmatrix}$$

From the looks of it, we can conclude that with the introduction of reversed polynomials

$$A^R(x) = x^n A(x^{-1}) = a_n + a_{n-1}x + \dots + a_0 x^n$$

$$B^R(x) = x^m B(x^{-1}) = b_m + b_{m-1}x + \dots + b_0 x^m$$

$$D^R(x) = x^{n-m} D(x^{-1}) = d_{n-m} + d_{n-m-1}x + \dots + d_0 x^{n-m}$$

the system may be rewritten as

$$A^R(x) \equiv B^R(x) D^R(x) \pmod{x^{n-m+1}}.$$

From this you can unambiguously recover all coefficients of $D(x)$:

$$\boxed{D^R(x) \equiv A^R(x)(B^R(x))^{-1} \pmod{x^{n-m+1}}}$$

And from this, in turn, you can recover $R(x)$ as $R(x) = A(x) - B(x)D(x)$.

Note that the matrix above is a so-called triangular Toeplitz matrix and, as we see here, solving system of linear equations with arbitrary Toeplitz matrix is, in fact, equivalent to polynomial inversion. Moreover, inverse matrix of it would also be triangular Toeplitz matrix and its entries, in terms used above, are the coefficients of $(B^R(x))^{-1} \pmod{x^{n-m+1}}$.

Calculating functions of polynomial

Newton's method

Let's generalize the Sieveking–Kung algorithm. Consider equation $F(P) = 0$ where $P(x)$ should be a polynomial and $F(x)$ is some polynomial-valued function defined as

$$F(x) = \sum_{i=0}^{\infty} \alpha_i (x - \beta)^i,$$

where β is some constant. It can be proven that if we introduce a new formal variable y, we can express $F(x)$ as

$$F(x) = F(y) + (x - y)F'(y) + (x - y)^2 G(x, y),$$

where $F'(x)$ is the derivative formal power series defined as

$$F'(x) = \sum_{i=0}^{\infty} (i+1)\alpha_{i+1}(x - \beta)^i,$$

and $G(x, y)$ is some formal power series of x and y. With this result we can find the solution iteratively.

Let $F(Q_k) \equiv 0 \pmod{x^a}$. We need to find $Q_{k+1} \equiv Q_k + x^a C \pmod{x^{2a}}$ such that $F(Q_{k+1}) \equiv 0 \pmod{x^{2a}}$.

Substituting $x = Q_{k+1}$ and $y = Q_k$ in the formula above, we get

$$F(Q_{k+1}) \equiv F(Q_k) + (Q_{k+1} - Q_k)F'(Q_k) + (Q_{k+1} - Q_k)^2 G(x, y) \pmod{x}^{2a}.$$

Since $Q_{k+1} - Q_k \equiv 0 \pmod{x^a}$, it also holds that $(Q_{k+1} - Q_k)^2 \equiv 0 \pmod{x^{2a}}$, thus

$$0 \equiv F(Q_{k+1}) \equiv F(Q_k) + (Q_{k+1} - Q_k)F'(Q_k) \pmod{x^{2a}}.$$

The last formula gives us the value of Q_{k+1}:

$$\boxed{Q_{k+1} = Q_k - \frac{F(Q_k)}{F'(Q_k)} \pmod{x^{2a}}}$$

Thus, knowing how to invert polynomials and how to compute $F(Q_k)$, we can find n coefficients of P with the complexity

$$T(n) = T(n/2) + f(n),$$

where $f(n)$ is the time needed to compute $F(Q_k)$ and $F'(Q_k)^{-1}$ which is usually $O(n \log n)$.

The iterative rule above is known in numerical analysis as Newton's method.

Hensel's lemma

As was mentioned earlier, formally and generically this result is known as Hensel's lemma and it may in fact used in even broader sense when we work with a series of nested rings. In this particular case we worked with a sequence of polynomial remainders modulo x, x^2, x^3 and so on.

Another example where Hensel's lifting might be helpful are so-called p-adic numbers where we, in fact, work with the sequence of integer remainders modulo p, p^2, p^3 and so on. For example, Newton's method can be used to find all possible automorphic numbers (numbers that end on itself when squared) with a given number base. The problem is left as an exercise to the reader. You might consider this problem to check if your solution works for 10-based numbers.

Logarithm

For the function $\ln P(x)$ it's known that:

$$\boxed{(\ln P(x))' = \frac{P'(x)}{P(x)}}$$

Thus we can calculate n coefficients of $\ln P(x)$ in $O(n \log n)$.

Inverse series

Turns out, we can get the formula for A^{-1} using Newton's method. For this we take the equation $A = Q^{-1}$, thus:

$$F(Q) = Q^{-1} - A$$

$$F'(Q) = -Q^{-2}$$

$$\boxed{Q_{k+1} \equiv Q_k(2 - AQ_k) \pmod{x^{2^{k+1}}}}$$

Exponent

Let's learn to calculate $e^{P(x)} = Q(x)$. It should hold that $\ln Q = P$, thus:

$$F(Q) = \ln Q - P$$

$$F'(Q) = Q^{-1}$$

$$\boxed{Q_{k+1} \equiv Q_k(1 + P - \ln Q_k) \pmod{x^{2^{k+1}}}}$$

k-th power

Now we need to calculate $P^k(x) = Q$. This may be done via the following formula:

$$Q = \exp\left[k \ln P(x)\right]$$

Note though, that you can calculate the logarithms and the exponents correctly only if you can find some initial Q_0.

To find it, you should calculate the logarithm or the exponent of the constant coefficient of the polynomial.

But the only reasonable way to do it is if $P(0) = 1$ for $Q = \ln P$ so $Q(0) = 0$ and if $P(0) = 0$ for $Q = e^P$ so $Q(0) = 1$.

Thus you can use formula above only if $P(0) = 1$. Otherwise if $P(x) = \alpha x^t T(x)$ where $T(0) = 1$ you can write that:

$$\boxed{P^k(x) = \alpha^k x^{kt} \exp[k \ln T(x)]}$$

Note that you also can calculate some k-th root of a polynomial if you can calculate $\sqrt[k]{\alpha}$, for example for $\alpha = 1$.

Evaluation and Interpolation

Chirp-z Transform

For the particular case when you need to evaluate a polynomial in the points $x_r = z^{2r}$ you can do the following:

$$A(z^{2r}) = \sum_{k=0}^{n} a_k z^{2kr}$$

Let's substitute $2kr = r^2 + k^2 - (r-k)^2$. Then this sum rewrites as:

$$\boxed{A(z^{2r}) = z^{r^2} \sum_{k=0}^{n} (a_k z^{k^2}) z^{-(r-k)^2}}$$

Which is up to the factor z^{r^2} equal to the convolution of the sequences $u_k = a_k z^{k^2}$ and $v_k = z^{-k^2}$.

Note that u_k has indexes from 0 to n here and v_k has indexes from $-n$ to m where m is the maximum power of z which you need.

Now if you need to evaluate a polynomial in the points $x_r = z^{2r+1}$ you can reduce it to the previous task by the transformation $a_k \to a_k z^k$.

It gives us an $O(n \log n)$ algorithm when you need to compute values in powers of z, thus you may compute the DFT for non-powers of two.

Another observation is that $kr = \binom{k+r}{2} - \binom{k}{2} - \binom{r}{2}$. Then we have

$$\boxed{A(z^r) = z^{-\binom{r}{2}} \sum_{k=0}^{n} \left(a_k z^{-\binom{k}{2}}\right) z^{\binom{k+r}{2}}}$$

The coefficient of x^{n+r} of the product of the polynomials $A_0(x) = \sum\limits_{k=0}^{n} a_{n-k} z^{-\binom{n-k}{2}} x^k$ and $A_1(x) = \sum\limits_{k\geq 0} z^{\binom{k}{2}} x^k$ equals $z^{\binom{r}{2}} A(z^r)$.
You can use the formula $z^{\binom{k+1}{2}} = z^{\binom{k}{2}+k}$ to calculate the coefficients of $A_0(x)$ and $A_1(x)$.

Multi-point Evaluation

Assume you need to calculate $A(x_1), \dots, A(x_n)$. As mentioned earlier, $A(x) \equiv A(x_i) \pmod{x - x_i}$. Thus you may do the following:

1. Compute a segment tree such that in the segment $[l, r)$ stands the product $P_{l,r}(x) = (x - x_l)(x - x_{l+1}) \dots (x - x_{r-1})$.
2. Starting with $l = 1$ and $r = n + 1$ at the root node. Let $m = \lfloor (l + r)/2 \rfloor$. Let's move down to $[l, m)$ with the polynomial $A(x) \pmod{P_{l,m}(x)}$.
3. This will recursively compute $A(x_l), \dots, A(x_{m-1})$, now do the same for $[m, r)$ with $A(x) \pmod{P_{m,r}(x)}$.
4. Concatenate the results from the first and second recursive call and return them.

The whole procedure will run in $O(n \log^2 n)$.

Interpolation

There's a direct formula by Lagrange to interpolate a polynomial, given set of pairs (x_i, y_i):

$$\boxed{A(x) = \sum_{i=1}^{n} y_i \prod_{j \neq i} \frac{x - x_j}{x_i - x_j}}$$

Computing it directly is a hard thing but turns out, we may compute it in $O(n \log^2 n)$ with a divide and conquer approach:

Consider $P(x) = (x - x_1) \dots (x - x_n)$. To know the coefficients of the denominators in $A(x)$ we should compute products like:

$$P_i = \prod_{j \neq i} (x_i - x_j)$$

But if you consider the derivative $P'(x)$ you'll find out that $P'(x_i) = P_i$. Thus you can compute P_i's via evaluation in $O(n \log^2 n)$.

Now consider the recursive algorithm done on same segment tree as in the multi-point evaluation. It starts in the leaves with the value $\frac{y_i}{P_i}$ in each leaf.

When we return from the recursion we should merge the results from the left and the right vertices as $A_{l,r} = A_{l,m} P_{m,r} + P_{l,m} A_{m,r}$.

In this way when you return back to the root you'll have exactly $A(x)$ in it. The total procedure also works in $O(n \log^2 n)$.

GCD and Resultants

Assume you're given polynomials $A(x) = a_0 + a_1 x + \dots + a_n x^n$ and $B(x) = b_0 + b_1 x + \dots + b_m x^m$.

Let $\lambda_0, \dots, \lambda_n$ be the roots of $A(x)$ and let $\mu_0, \dots, \mu_m$ be the roots of $B(x)$ counted with their multiplicities.

You want to know if $A(x)$ and $B(x)$ have any roots in common. There are two interconnected ways to do that.

Euclidean algorithm

Well, we already have an article about it. For an arbitrary domain you can write the Euclidean algorithm as easy as:

```
template<typename T>
T gcd(const T &a, const T &b) {
    return b == T(0) ? a : gcd(b, a % b);
}
```

It can be proven that for polynomials $A(x)$ and $B(x)$ it will work in $O(nm)$.

Resultant

Let's calculate the product $A(\mu_0) \cdots A(\mu_m)$. It will be equal to zero if and only if some μ_i is the root of $A(x)$.

For symmetry we can also multiply it with b_m^n and rewrite the whole product in the following form:

$$\boxed{\mathcal{R}(A, B) = b_m^n \prod_{j=0}^{m} A(\mu_j) = b_m^n a_m^n \prod_{i=0}^{n} \prod_{j=0}^{m} (\mu_j - \lambda_i) = (-1)^{mn} a_n^m \prod_{i=0}^{n} B(\lambda_i)}$$

The value defined above is called the resultant of the polynomials $A(x)$ and $B(x)$. From the definition you may find the following properties:

1. $\mathcal{R}(A, B) = (-1)^{nm} \mathcal{R}(B, A)$.
2. $\mathcal{R}(A, B) = a_n^m b_m^n$ when $n = 0$ or $m = 0$.
3. If $b_m = 1$ then $\mathcal{R}(A - CB, B) = \mathcal{R}(A, B)$ for an arbitrary polynomial $C(x)$ and $n, m \geq 1$.
4. From this follows $\mathcal{R}(A, B) = b_m^{\deg(A) - \deg(A - CB)} \mathcal{R}(A - CB, B)$ for arbitrary $A(x)$, $B(x)$, $C(x)$.

Miraculously it means that resultant of two polynomials is actually always from the same ring as their coefficients!

Also these properties allow us to calculate the resultant alongside the Euclidean algorithm, which works in $O(nm)$.

```
template<typename T>
T resultant(poly<T> a, poly<T> b) {
    if(b.is_zero()) {
        return 0;
    } else if(b.deg() == 0) {
        return bpow(b.lead(), a.deg());
    } else {
        int pw = a.deg();
        a %= b;
        pw -= a.deg();
        base mul = bpow(b.lead(), pw) * base((b.deg() & a.deg() & 1) ? -1 : 1);
        base ans = resultant(b, a);
        return ans * mul;
    }
}
```

Half-GCD algorithm

There is a way to calculate the GCD and resultants in $O(n \log^2 n)$.

The procedure to do so implements a 2×2 linear transform which maps a pair of polynomials $a(x)$, $b(x)$ into another pair $c(x), d(x)$ such that $\deg d(x) \leq \frac{\deg a(x)}{2}$. If you're careful enough, you can compute the half-GCD of any pair of polynomials with at most 2 recursive calls to the polynomials which are at least 2 times smaller.

The specific details of the algorithm are somewhat tedious to explain, however you can find its implementation in the library, as `half_gcd` function.

After half-GCD is implemented, you can repeatedly apply it to polynomials until you're reduced to the pair of $\gcd(a,b)$ and 0.

Problems

- CodeChef - RNG
- CodeForces - Basis Change
- CodeForces - Permutant
- CodeForces - Medium Hadron Collider

Contributors:
jakobkogler (60.9%) adamant-pwn (36.25%) hieplpvip (0.81%) peltorator (0.61%) hly1204 (0.61%) algmyr (0.41%) tanmay-sinha (0.41%)