# RNBWROAD - Editorial

editorial fft generating_functions hard math polynomials may20 shisuko

shisuko 6h

## PROBLEM LINK:

Practice
Div-2 Contest
Div-1 Contest

*Author & Editorialist:* Cisco Ortega

*Tester :* Danya Smelskiy

## DIFFICULTY:

Hard

## PREREQUISITES:

Math, Generating Functions, FFT

## PROBLEM:

Consider an $m \times n$ grid where $(1,1)$ is the top-left corner and $(m,n)$ is the bottom-right corner. From some cell, we can travel to the cell directly below it or the cell directly to its right. We are given an array of $n$ distinct integers $a_1, a_2, \dots a_n$, and all cells in column $j$ have the value $a_j$. The weight of a path is the product of all cells on a given path.

For $Q$ different values of $k_i$, find the sum of the weights of all paths that begin at $(1,1)$ and end at $(k_i, n)$, taken modulo $7 \times 17 \times 2^{23} + 1$.

## QUICK EXPLANATION:

1. Use generating functions to find that the sum of the weights of all paths from $(1,1)$ to $(m,n)$ is equal to the coefficient of $x^{m+n-1}$ in $\dfrac{x^n \prod_j a_j}{\prod_j (1-a_j x)}$.This is equivalent to finding the coefficient of $x^{m-1}$ in $\dfrac{(-1)^n}{\prod_j (x - a_j^{-1})}$.

2. If $Q \le 10$,

- Notice that the answer for some $k_i$ is independent of $m$. So, if we have a method to compute the answer for some $m$, we can use it for any query.

- Use partial fraction decomposition to express this as $(-1)^n \sum_j \dfrac{K_j}{x - a_j^{-1}}$, where $K_j$ is the respective coefficient of the $j$th term. We can do this in $\mathcal{O}(n \log^2 n)$
- We re-express this as $(-1)^{n+1} \sum_j \dfrac{K_j a_j}{1 - a_j x}$. With geometric series' expansion, the coefficient of $x^{m-1}$ is thus $(-1)^{n+1} \sum_j K_j a_j^m$. We can evaluate this sum in $\mathcal{O}(n \log m)$.
- The final complexity over all queries is $\mathcal{O}(n \log^2 n + Qn \log m)$.

3. If $m \leq 10^5$,

- We can evaluate $\prod_j (x - a_j^{-1})$ in $\mathcal{O}(n \log^2 n)$
- We can find the first $m$ terms of its reciprocal in $\mathcal{O}(m \log m)$
- We can answer each query in $\mathcal{O}(1)$ since getting the first $m$ terms gives us the answers for all $1 \leq k \leq m$.
- The final complexity over all queries is $\mathcal{O}(n \log^2 n + m \log m + Q)$

## EXPLANATION:

The following tutorial can be considered a crash course on several different techniques that are used in computationally manipulating polynomials. It will be a real marathon. We will assume, as prerequisite, that the reader has knowledge of the following techniques,

- Computing the modular inverse $a^{-1} \bmod p$ for some prime $p$
- Multiplying two $n$ degree polynomials in $\mathcal{O}(n \log n)$ using the Fast Fourier Transform.
- Performing a Number Theoretic Transform by using a primitive root of unity of a prime modulus instead of a complex root of unity.

and the full list of techniques covered by this editorial are,

- Given the first $n$ terms of a power series of $f(x)$, compute the first $n$ terms of the power series of its reciprocal $\dfrac{1}{f(x)}$ in $\mathcal{O}(n \log n)$.
- Given two polynomials $f(x)$ and $g(x)$, where $\deg f = n$ and $\deg g < \deg f$, perform the polynomial long division $f = gq + r$ in $\mathcal{O}(n \log n)$.
- Given $n$ numbers $r_1, r_2, \ldots r_n$, perform the "binary-splitting" algorithm for computing $(x - r_1)(x - r_2) \ldots (x - r_n)$ in $\mathcal{O}(n \log^2 n)$.
- Given an $n$ degree polynomial $A(x)$, evaluate it at $n$ distinct points $r_1, r_2, \ldots r_n$ in $\mathcal{O}(n \log^2 n)$
- Given the rational expression $\dfrac{1}{(x - r_1)(x - r_2) \ldots (x - r_n)}$ where the $r_i$ are pairwise-distinct, find its partial fraction decomposition in $\mathcal{O}(n \log^2 n)$.

Each technique will be in its own labeled and mostly self-contained section. For flow of the editorial, the techniques will be discussed in roughly the opposite order that they were listed above.

### Finding the Formula

We begin with some preliminary observations.

- The answer for some $k_i$ is actually independent of $m$. So, for the rest of this editorial, we will assume that the query is $m$, since the same technique will also work if $k_i$ were any other number.
- Each path from $(1, 1)$ to $(m, n)$ passes through exactly $m + n - 1$ cells.
- Suppose you counted that you passed through $c_1$ cells in the first column, then through $c_2$ cells in the second column, and so on, then through $c_n$ cells in the final column, where $1 \le c_i \le m$ and $c_1 + c_2 + \dots c_n = m + n - 1.$
- Once you 'exit' a column by moving right, you will never re-enter that column again.
- Thus, there is a **one-to-one** relation between a path from $(1, 1)$ to $(m, n)$ and such a sequence of values $(c_1, c_2, \dots c_n)$.

Thus, the problem can be reduce to evaluating the sum,

$$\sum_{c_1+c_2+\cdots+c_n=m+n-1} a_1^{c_1} a_2^{c_2} \dots a_n^{c_n}$$

where $1 \le c_i \le m$. If you are familiar with manipulating generating functions, you will recognize that this is

$$= [x^{m+n-1}] \prod_{j=1}^{n} (a_j x + (a_j x)^2 + \cdots + (a_j x)^m)$$

$$= [x^{m+n-1}] \prod_{j=1}^{n} \sum_{k=1}^{m} (a_j x)^k$$

where we use the notation $[x^k]f(x)$ to mean "the coefficient of $x^k$ in $f(x)$". We can pull out the common $a_j x$ in each multiplicand to get

$$[x^{m+n-1}] \prod_{j=1}^{n} \left( a_j x \sum_{k=0}^{m-1} (a_j x)^k \right)$$

$$= [x^{m+n-1}] x^n \prod_{j=1}^{n} \left( a_j \sum_{k=0}^{m-1} (a_j x)^k \right)$$

Since multiplying by $x^n$ merely shifts all the terms in the final product $n$ places up, this is equivalent to finding the coefficient

$$[x^{m-1}] \prod_{j=1}^{n} a_j \sum_{k=0}^{m-1} (a_j x)^k$$

Now, this is a geometric series. If it were an *infinite* geometric series, it would have a neat formula, but unfortunately our series are only finite. However, notice that we only need the coefficient of $x^{m-1}$ in the final product. So, even if we add extra terms like $x^m$ or $x^{m+1}$ or higher, they will be too large to contribute to the coefficient of a lower-degree term. So, what we are looking for is in fact equal to

$$= [x^{m-1}] \prod_{j=1}^{n} a_j \sum_{k=0}^{\infty} (a_j x)^k$$

$$= [x^{m-1}] \prod_{j=1}^{n} \frac{a_j}{1 - a_j x}$$

How do we then find the coefficient of $x^{m-1}$ in this expression? If we had a simpler case with $n = 1$, then

$$[x^{m-1}] \frac{r}{1 - rx} = r^m$$

by the geometric series formula (since we are merely using this as a formal power series and won't actually evaluate the function at any point, we don't need to worry about convergence). This motivates us to recall a technique from high school—partial fraction decomposition. For reasons which will become clearer later, it will be more convenient to rewrite the above expression as

$$= [x^{m-1}] \prod_{j=1}^{n} \frac{-1}{x - a_j^{-1}}$$

$$= [x^{m-1}] (-1)^n \prod_{j=1}^{n} \frac{1}{x - a_j^{-1}}$$

With partial fraction decomposition, we can rewrite the above product as

$$= [x^{m-1}] (-1)^n \sum_{j=1}^{n} \frac{K_j}{x - a_j^{-1}}$$

where each $K_j$ is some constant coefficient. If we are able to find such $K_j$ for the decomposition, then the answer we want is simply going to be

$$= (-1)^n \sum_{j=1}^{n} [x^{m-1}] \frac{K_j}{x - a_j^{-1}}$$

$$= (-1)^{n+1} \sum_{j=1}^{n} [x^{m-1}] K_j \frac{a_j}{1 - a_j x}$$

$$= (-1)^{n+1} \sum_{j=1}^{n} K_j a_j^m$$

So, the question is—how do we efficiently perform a partial fraction decomposition?

## Partial Fraction Decomposition

Suppose we want to perform the partial fraction decomposition,

$$\prod_{i=1}^{n} \frac{1}{x - r_i} = \sum_{i=1}^{n} \frac{K_i}{x - r_i}$$

where the $r_i$ are distinct (in our problem, these would be the $a_i^{-1}$). Multiply both sides by $\prod(x - r_i)$ to get

$$1 = \sum_{i=1}^{n} K_i \prod_{j \neq i} (x - r_j)$$

Since the $r_i$ are distinct, we can solve for a particular $K_k$ by plugging in $x = r_k$ into the equation. For every term except for when $i = k$, notice that $\prod_{j \neq i}(r_k - r_j) = 0$. So, we are ultimately left with

$$K_k = \left( \prod_{j \neq k} (r_k - r_j) \right)^{-1}$$

If we were to naively compute this product for each $1 \leq k \leq n$, this would take $\mathcal{O}(n^2)$ time, which is too slow. So, we have to employ a smarter trick. I attribute the following technique to the paper "Complexity of numerical algorithms for polynomials" (Chin, 1977). Define

$$Q(x) = \prod_{j=1}^{n} (x - r_j)$$

Let's try to take the derivative of $Q(x)$. We compute for $Q'(x)$ using the product rule,

$$\begin{aligned} Q'(x) &= \sum_{i=1}^{n} \left( \prod_{j \neq i} (x - r_j) \right) \frac{\mathrm{d}}{\mathrm{d}x} (x - r_i) \\ &= \sum_{i=1}^{n} \prod_{j \neq i} (x - r_j) \end{aligned}$$

Notice something neat? What happens when you plug in $x = r_k$? Similar to the trick shown earlier, all the terms vanish except for when $i = k$. So, we get

$$Q'(r_k) = \prod_{j \neq k} (r_k - r_j)$$

which leads us to the amazing realization that

$$K_k = (Q'(r_k))^{-1}$$

So, to find each of the $K_k$, all we have to do is evaluate the polynomial $Q'(x)$ at the $n$ different points $r_1, r_2, \ldots r_n$. So, we now have the following two-step plan.

- Find the explicit coefficients of $Q'(x)$
- Evaluate $Q'(x)$ at $n$ different points in subquadratic time.

## Finding $Q'(x)$ with Binary Splitting

If we knew the coefficients of

$$Q(x) = \sum_{k=0}^{n} q_k x^k$$

then the coefficients of $Q'(x)$ will simply be

$$Q'(x) = \sum_{k=0}^{n-1} (k+1) q_{k+1} x^k$$

So, it suffices for us to be able to find the coefficients of $Q(x)$. Recall that we defined $Q(x)$ to be

$$Q(x) = (x - r_1)(x - r_2) \ldots (x - r_n)$$

If we were to multiply each $x - r_i$ one-by-one from left to right, we would get an $\mathcal{O}(n^2)$ time complexity. So, we will do something smarter instead—Divide and Conquer.

Let $M = \lfloor \frac{1+n}{2} \rfloor$. Then, we define two functions

$$Q_1(x) = (x - r_1)(x - r_2) \ldots (x - r_M)$$

$$Q_2(x) = (x - r_{M+1})(x - r_{M+2}) \ldots (x - r_n)$$

We notice that computing the coefficients of each of these is a smaller subproblem. How much smaller are the subproblems? Note that $\deg Q_1 \approx \frac{n}{2}$. Similarly, $\deg Q_2 \approx \frac{n}{2}$.

Also, once we have the coefficients of $Q_1(x)$ and $Q_2(x)$, we can merge their answers together to get the coefficients of $Q(x)$ since $Q(x) = Q_1(x) Q_2(x)$. This gives us a recurrence of

$$T(n) = 2T\left(\frac{n}{2}\right) + \mathcal{O}(n \log n)$$

if we use FFT (an NTT, to be precise) to multiply $Q_1(x)$ and $Q_2(x)$ together. By Master Theorem, this has a running time of $T(n) = \mathcal{O}(n \log^2 n)$. I have heard this technique be referred to as **binary splitting** in the literature.

## Evaluating $Q'(x)$ at $n$ arbitrary points

Fun fact: This is given as a guided exercise in CLRS!

Suppose we have some $n$ degree polynomial $A(x)$ which we need to evaluate at the $n$ different points $r_1, r_2, \ldots r_n$. For our problem, we will be evaluating $Q'(x)$, but this technique will work for any $n$ degree polynomial. To do so, we will employ our favorite vanishing trick in order to perform some divide and conquer.

Consider the $Q_1$ and $Q_2$ from the last section. Consider some $r_k$ where $1 \leq k \leq M$. Due to how it was defined, $Q_1(r_k) = 0$. Similarly, if $M + 1 \leq k \leq n$, then $Q_2(r_k) = 0$. This motivates the following steps.

Notice that the degrees of $Q_1$ and $Q_2$ are less than $\deg Q' = n - 1$. So, we can use polynomial long division to write

$$A(x) = Q_1(x)B_1(x) + A_1(x)$$

and

$$A(x) = Q_2(x)B_2(x) + A_2(x)$$

where $A_1$ and $A_2$ are the unique polynomials such that $\deg A_1 < \deg Q_1$ and $\deg A_2 < \deg Q_2$.

What happens then if plug in each $r_k$ into $A(x)$? Note that if $1 \le k \le M$, since $Q_1(r_k) = 0$ ,

$$A(r_k) = A_1(r_k)$$

and if $M + 1 \le k \le n$, since $Q_2(r_k) = 0$,

$$A(r_k) = A_2(r_k)$$

In summary, $A(x)$ can be defined on the $n$ points as

$$A(r_k) = \begin{cases} A_1(r_k) & \text{if } 1 \le k \le M \\ A_2(r_k) & \text{if } M+1 \le k \le n \end{cases}$$

where $A_1$ can be thought of as $A \bmod Q_1$, and $A_2$ can be thought of as $A \bmod Q_2$.

So, in order to evaluate $A(x)$ at each of the $r_k$, we can evaluate $A_1(x)$ at $r_1, r_2, \ldots r_M$, and evaluate $A_2(x)$ at $r_{M+1}, r_{M+2}, \ldots r_n$. These are smaller subproblems, so we apply Divide and Conquer. We are assured that $\deg A_1 < \frac{n}{2}$ and $\deg A_2 < \frac{n}{2}$ (why?)

How do we find $Q_1$ and $Q_2$? Well, we already computed them earlier when we were performing binary splitting! We should just modify our binary splitting algorithm so that it actually stores the coefficients of each $Q_1$ and $Q_2$ somewhere at each step of the process.

Of course, there is the actual matter of actually *finding* $A_1(x)$ and $A_2(x)$. We claim for now that polynomial long division can be performed in $\mathcal{O}(n \log n)$.

Thus, we have a recurrence of $T(n) = 2T(\frac{n}{2}) + \mathcal{O}(n \log n)$. Again, by Master Theorem, that means our algorithm has a complexity of $\mathcal{O}(n \log^2 n)$.

So, how do we perform polynomial long division in $\mathcal{O}(n \log n)$?

## Polynomial Long Division

Suppose we have two polynomials $f$ and $g$ such that $\deg g < \deg f$. We wish to find the unique polynomials $q$ and $r$ such that $f = gq + r$ and $\deg r < \deg g$.

Performing synthetic division will unfortunately be $O(n^2)$ (if $n = \deg f$). We will need somethig smarter. There are many ways to do polynomial long division, but the following method is what I found to be the easiest to understand, taken from some lecture notes from MIT .

What if, miraculously, $r(x) = 0$? Then, we would have $f = gq$. From here, it seems natural that the next step is to 'divide' both sides by $g$, as if this were just normal algebra. It turns out that this is actually something we can do.

For some polynomial $g$, let's define its reciprocal $\frac{1}{g}$ to be the polynomial such that $g\frac{1}{g} = 1$. Note here that $1$ refers to a function, the constant function that always returns 1, which is the identity in polynomial multiplication. Note that oftentimes the reciprocal is an *infinite* polynomial. We will later discuss when such a reciprocal exists and how to compute it, but for now let us take it on faith that this is something we can do.

From $f = gq$, multiply both sides by $\frac{1}{g}$ and get $\frac{1}{g}f = \frac{1}{g}gq = 1q = q$. This motivates us to introduce some definitions to make the $r(x)$ vanish from the equation.

**Definition:** We introduce a new operator $rev(f)$, which we read as the "reverse" of $f$. Simply put, we 'reverse' the order of the coefficients of $f$. So, if

$$f(x) = \sum_{k=0}^{n} a_k x^k$$

then

$$rev(f)(x) = \sum_{k=0}^{n} a_k x^{n-k}$$

**Exercise:** Prove that $rev(fg) = rev(f)rev(g)$ (aka the proof is by straightforward algebraic manipulations which I don't feel like typing out).

This turns out to be pretty important, because we then have

$$rev(f) = rev(gq + r) = rev(g)rev(q) + \sum_{k=0}^{\deg r} c_k x^{\deg f - k}$$

if $c_k$ are the coefficients of $r(x)$. Note that $rev(gq + r) \neq rev(g)rev(q) + rev(r)$ because we have to reverse $r$ with respect to the larger $\deg f$.

Now, by the definition of polynomial long division, $\deg r < \deg g$. So, $\deg f - \deg r > \deg f - \deg g$. The degrees of the terms in the sum on the right are thus rather large—each has degree at least $\deg f - \deg g + 1$. We can make them vanish with the following definition.

**Definition:** Let $f$ and $g$ be polynomials. We say that $f \equiv g \bmod x^m$ if the coefficients of $f$ and $g$ agree at the first $m$ terms. Since we start counting from 0, that means their coefficients must be equal from $x^0$ to $x^{m-1}$.

Since all the terms in that sum on the right are $\deg f - \deg g + 1$ or larger, we have

$$rev(f) \equiv rev(g)rev(q) \mod x^{\deg f - \deg g + 1}$$

and so we are free to do

$$rev(q) \equiv rev(f)\frac{1}{rev(g)} \mod x^{\deg f - \deg g + 1}$$

where $\frac{1}{rev(g)}$ is the polynomial such that $rev(g)\frac{1}{rev(g)} \equiv 1 \bmod x^{\deg f - \deg g + 1}$. Recall earlier that we said that the reciprocal could possibly be an infinite polynomial. However, we do not need all infinitely many terms—knowing its first few terms will already suffice, since we are working with a modulo. We will expound more on how to actually find this reciprocal later.

Now, note that $\deg q = \deg f - \deg g$. This means that $q$ (and thus $rev(q)$) have no terms with degree $\deg f - \deg g + 1$ or greater. Thus, if we compute

$$rev(f)\frac{1}{rev(g)} \mod x^{\deg f - \deg g + 1}$$

then the result that we get is **exactly** $rev(q)$. We can reverse this again to recover $q$.

Again, if $\deg f = n$, then each reverse can be done in $\mathcal{O}(n)$. I claim that $\frac{1}{rev(g)} \bmod x^{\deg f - \deg g + 1}$ can be computed in $\mathcal{O}(n \log n)$. Finally, the product $rev(f)\frac{1}{rev(g)} \bmod x^{\deg f - \deg g}$ can be computed in $\mathcal{O}(n \log n)$ as well using FFT.

So, how do we compute $\frac{1}{rev(g)} \bmod x^{\deg f - \deg g}$?

## Finding a reciprocal $\bmod x^n$

Given some the first $n$ terms of some polynomial $G(x)$, we say that $R_n(x) = \frac{1}{G(x)} \bmod x^n$ if $R_n(x)$ is the polynomial such that

$$G(x)R_n(x) \equiv 1 \mod x^n$$

Note that the 1 there is the constant function $f(x) = 1$. I claim that such an $R_n(x)$ exists for all $G(x)$ such that $G(0) \neq 0$.

I first encountered the following technique from the editorial of the Codeforces problem The Child and Binary Tree . The technique is reminiscent of 'lifting' algorithms. We will find a solution in some modulus, and find a way to 'lift' that solution up to a greater mod.

In particular, if we know $R_k(x)$, our algorithm will produce for us the function $R_{2k}(x)$.

**Exercise**: Find the value of $R_1(x)$. **Hint**: It's constant.

Suppose we have some $R_k(x)$ such that

$$R_k(x)G(x) \equiv 1 \mod x^k$$

$$R_k(x)G(x) - 1 \equiv 0 \mod x^k$$

Since here, the term of smallest degree with a nonzero coefficient is at least $x^k$, when we square it,

$$(R_k(x)G(x) - 1)^2 \equiv 0 \mod x^{2k}$$

we are certain that the term of smallest degree with a nonzero coefficient is now at least $x^{2k}$. Expanding, we get

$$R_k(x)^2G(x)^2 - 2R_k(x)G(x) + 1 \equiv 0 \mod x^{2k}$$

$$1 \equiv 2R_k(x)G(x) - R_k(x)^2G(x)^2 \mod x^{2k}$$

$$1 \equiv (2R_k(x) - R_k(x)^2G(x))G(x) \mod x^{2k}$$

By the definition then, we see that

$$R_{2k}(x) \equiv 2R_k(x) - R_k(x)^2G(x) \mod x^{2k}$$

Since we can easily procure the starting point $R_1(x)$, we can iterate this algorithm as many times as necessary until we reach a power of 2 that is at least $n$. If $2^k \geq n$, then $R_{2^k}(x) \equiv R_n(x) \mod x^n$ (why?).

What is the runtime of this algorithm? At each step, we have to perform a constant number of multiplications. Crucially, note that computing $R_k(x)$ only requires multiplying the first $k$ terms of all involved polynomials (since it will be reduced $\mathrm{mod}\ x^k$ anyway), so we only perform $\mathcal{O}(k \log k)$ operations at each step.

Thus, we have a running time of $T(n) = T(\frac{n}{2}) + \mathcal{O}(n \log n)$, which by Master Theorem has a complexity of $\mathcal{O}(n \log n)$.

**Fun Exercise:** Use this idea to create an algorithm which finds the first $n$ terms of $\sqrt{G(x)}$ in $\mathcal{O}(n \log^2 n)$.

## In summary

- We can multiply two polynomials in $\mathcal{O}(n \log n)$ using FFT
- With this, we can find the reciprocal of a polynomial $\mathrm{mod}\ x^n$ in $\mathcal{O}(n \log n)$ using a lifting algorithm.
- With this, we can do polynomial long division in $\mathcal{O}(n \log n)$ using the reversing trick.
- We can evaluate $(x - r_1)(x - r_2)\dots(x - r_n)$ in $\mathcal{O}(n \log^2 n)$ using binary splitting.
- With these, we can evaluate a degree $n$ polynomial at $n$ arbitrary points in $\mathcal{O}(n \log^2 n)$ using a Divide and Conquer algorithm.
- With this, we can compute the partial fraction decomposition of $n$ different linear terms in $\mathcal{O}(n \log^2 n)$ with Chin's derivative trick.
- With this, we can find the answer to one query in $\mathcal{O}(n \log m)$

## What if there are a lot of queries?

If we used the above algorithm to answer all queries, then—including preprocessing—we would have a time complexity of $\mathcal{O}(n \log^2 n + Qn \log m)$, which will pass the third and fourth subtasks. However, this actually won't pass the second (and possibly even the first!) subtask because there are too many queries.

However, notice that in the first and second subtasks, $m \le 10^5$. For these cases, we don't even need to bother with the partial fraction decomposition at all!

Recall that what we want to find is the coefficient of $x^{m-1}$ in $\frac{(-1)^n}{Q(x)}$. What you could do instead then is compute $Q(x)$ in $\mathcal{O}(n \log^2 n)$ using binary splitting, then find the first $m$ terms of $\frac{1}{Q(x)}$ in $\mathcal{O}(m \log m)$ with the reciprocal-finding algorithm.

Note that this *already* produces the coefficients of the first $m$ terms, so we already have the answer for all $k$ in $1 \le k \le m$, all at once.

With this solution, we can answer all queries in $\mathcal{O}(1)$ because it becomes a simple array lookup. This method gives a complexity of $\mathcal{O}(n \log^2 n + m \log m + Q)$, which will pass the first two subtasks.

We can combine this with our first solution with an if statement that checks if $m$ is small enough to do this.

### Final Words

If you made it this far, congratulations! I hope you learned something new about polynomials. This editorial is more comprehensive and elaborate because I wanted there to be a good reference for these techniques. I wanted to give something back to the competitive programming community, since I had learned so much from the generous blog posts of others.

I love how each of the techniques builds on each other, each with its own interesting trick. I hope to see more generating function and polynomial manipulation problems in contests!

The original version of this problem had $a_j = j$, i.e. $\{a_n\} = 1, 2, 3, \ldots n$. Its solution noted a relationship with *Stirling Numbers of the Second Kind* in order to solve it sub-quadratically. Alei asked me to make the problem more algorithmic, so I changed it to the current version of the problem.

When I first wrote this problem, I did not know how to do partial fractions or polynomial long division—I only learned those with the help of Google. I also theoretically knew about multi-point evaluation due to CLRS but this was my first time implementing it. This was just as much a learning experience for me, but it seems that a lot of 7☆ participants already had this prepared in their libraries!

Thanks for reading, and I hope you enjoyed this editorial :))

## SOLUTIONS:

► Setter's Solution

psaini72 3h

I really look forward to questions that involve generating functions and FFT in long challenges, but I was kind of disappointed this time. The last 2 subtasks (partial fractions) of this problem are a subpart of CHEFEQUA . After getting the generating function (which is fairly easy to get ), the solution is nothing new if you have solved CHEFEQUA.

Having said this, I really want to congratulate the setter for creating this problem on their own. The problem is genuinely hard and took me a lot of time to solve, and I really enjoyed solving it … in November 2018. However I'm sure that I was only able to solve it because I knew there had to be a solution, as it was in a contest. As a setter, you cannot know if there even is an efficient solution or not. Well done for sticking with the problem long enough to discover the solution.

Great job on the editorial too. I think that the only resource on multi-point evaluation using FFT was just a loose description of it on stackexchange. I had to spend some time to fill in some details when I first saw it. Similar case for efficient quotient and modulo, there were some densely written class notes from some college that I had to decipher.

As far as the first 2 subtasks go (calculating the generating function by inverting), that is the first thing you can do with FFT after just multiplying 2 polynomials. The question would've been better off without that part IMO.

---

bohdan 2h

Nice & hard problem!
This article help me a lot: https://cp-algorithms.com/algebra/polynomial.html

---

carre 2h

I'm not sure I can understand what you're trying to say. Perhaps my language limitations or a limited ability to interpret sarcasm. Are you saying you encountered this problem (similar?) previously in a contest? You have solved it previusly but not a contest?
If that's the case, I have no idea about this particular problem, but overall it's not that rare that there have been similar problems in the past.
Sorry if I misunderstood your point.

---

psaini72 1h

There is no sarcasm. I genuinely believe that the setter has done a good job.

> carre:
>
> Are you saying you encountered this problem (similar?) previously in a contest?

I'm saying that I saw an almost Identical problem in NOV18A on Codechef.

> carre:

> but overall it's not that rare that there have been similar problems in the past

I have never seen 2 problems on codechef that involve FFT but use the *exact* same idea. And there have been around 10 problems in Long Challenges that require FFT.

I get that concepts are reused across problems, and that it's not the end of the world, just that I'm a bit disappointed.

edit: @carre I realise how horribly it's written. Rearranged a few paragraphs.

---

shisuko 8m

Thanks for the feedback! Don't worry, I didn't find your comment to be sarcastic. It's very much appreciated, in fact.

I only started using CodeChef last year so I didn't know about that CHEFEQUA problem. Oh well, I'm a bit disappointed too, but nothing we can do about that now. The low solve-rate of RNBWROAD in spite of that at least shows that this did not ruin the contest, which I am thankful for.

Considering this fact, I'm actually surprised the solve rate isn't higher, since this problem is admittedly rather standard (finding the generating function, like you said, isn't too hard) and most of the difficulty is in knowing how to implement each of these techniques. Once again, oh well. Hopefully this editorial, as a resource, leads to more people learning these nice techniques.

The first two and last two subtasks were actually different versions that I proposed as a more general version of the original problem I came up with. I made the call to merge them into one problem. It was considered to make them different problems, like BIPPAIR and PRECPAIR, but I didn't feel like the two versions for my problem were sufficiently different to warrant this. In fact, getting the reciprocal of a polynomial is needed for polynomial long division, so the prerequisite techniques for the big Q case are a strict subset of the techniques for the big M case. It's rather inelegant, but I still think it was still a subtask worth having. I admit it's debatable, though.

Again, thank you for your feedback! I hope we see more FFT and generating functions too in Long Challenges to come :))