**Fast Fourier Transform and its applications**

First, let’s define a problem: you are given two polynomials A(x) and B(x). Your task is to compute A(x)B(x) in reasonable amount of time.

Of course we have a straightforward solution that runs in time, but it’s too slow.

At first we won’t focus on our initial problem: now we’re going to discuss complex numbers. As you know by Euler’s formula . Using this fact we can find all the solutions to equation

We know that .

From here we can see that — roots of unity. I’ll use the following designation: , note that the k-th root of unity is just .

So now suppose you have a task to evaluate some polynomial at the roots of unity.

From now on I’ll assume that for some natural number k, if it’s not, then just append zeros to our initial polynomial.

Let’s use the divide & conquer method: we split our polynomial into two polynomials:

;

Now one can easily see that

Now notice these interesting properties of roots of unit

* , notice that

By this we can easily get the formula for :

. But this only works if . If it’s not, the and also .

Now comes the divide & conquer approach, to calculate P(x), we calculate A(x) and B(x), so

After all these observations we can see that:

* , if
* , if

So the total complexity of this algorithm will be . But still by now we can do nothing with multiplying two polynomials. So we probably need the inverse Fourier transform.

I’m not going to go into math details, but all you have to do in order to compute the inverse discrete Fourier transform ( restore polynomial knowing its DFT ) is at first: set

. And at second, after you run simple FFT using new roots of unity, you need to divide all the coefficients by .

Going back to the initial problem: all we have to do in order to multiply two polynomials we need to double their sizes (because we want to multiply them, and they will get much bigger), calculate FFF(A), FFT(B), then multiply the coefficients at corresponding positions and compute the INVERSE\_FFT(A) – it will be exactly .

Some good tasks involving FFT: [632E](https://codeforces.com/problemset/problem/632/E), [958F3](https://codeforces.com/problemset/problem/958/F3).

**Number theoretic transform**

FFT works great, but the problem is that it uses double numbers, which might have big precision errors. In order to get rid of them, we still want to multiply two polynomials, but compute every coefficient modulo some prime number p. For small modulo this can be done using casual FFT, but if , then FFT won’t work normally.

Thus, we discovered another way of doing FFT. We won’t use roots of one anymore. Instead, we’ll use the modular arithmetic roots of unity. At first, let’s define n-th root of unity as follows:

I claim that exists only if our prime modulo is in the form , and . If these two conditions are satisfied and , then

So our task is to evaluate some polynomial at these points: . Let’s use divide & conquer approach again: split P(x) into A(x) and B(x):

,

Now notice some interesting facts: , thus is root of unity. .

So now . Now if , then , and if it’s not, then , . .

Thus .

And the final algorithm for NTT will be:

1. Divide out polynomial P(x) into A(x) and B(x)
2. Calculate roots of unity
3. Run NNT(A), NTT(B):
4. For every

Time complexity is also .

My implementation of [NTT](https://github.com/vovmr/All-algorithms-for-competetive-programming/blob/master/good%20ntt.cpp), [FFT](https://github.com/vovmr/All-algorithms-for-competetive-programming/blob/master/FFT%20for%20958F3.cpp). More information can be found [here](https://cp-algorithms.com/algebra/fft.html) and [here (in russian)](https://habr.com/ru/post/113642/).