Fast Fourier transform

Table of Contents

In this article we will discuss an algorithm that allows us to multiply two polynomials of length n in $O(n \log n)$ time, which is better than the trivial multiplication which takes $O(n^2)$ time. Obviously also multiplying two long numbers can be reduced to multiplying polynomials, so also two long numbers can be multiplied in $O(n \log n)$ time (where n is the number of digits in the numbers).

The discovery of the **Fast Fourier transformation (FFT)** is attributed to Cooley and Tukey, who published an algorithm in 1965. But in fact the FFT has been discovered repeatedly before, but the importance of it was not understood before the inventions of modern computers. Some researchers attribute the discovery of the FFT to Runge and König in 1924. But actually Gauss developed such a method already in 1805, but never published it.

Discrete Fourier transform

Let there be a polynomial of degree $n-1$:

$$A(x) = a_0 x^0 + a_1 x^1 + \cdots + a_{n-1} x^{n-1}$$

Without loss of generality we assume that n - the number of coefficients - is a power of 2. If n is not a power of 2, then we simply add the missing terms $a_i x^i$ and set the coefficients a_i to 0.

The theory of complex numbers tells us that the equation $x^n = 1$ has n complex solutions (called the n-th roots of unity), and the solutions are of the form $w_{n,k} = e^{\frac{2k\pi i}{n}}$ with $k = 0 \ldots n-1$. Additionally these complex numbers have some very interesting properties: e.g. the principal n-th root $w_n = w_{n,1} = e^{\frac{2\pi i}{n}}$ can be used to describe all other n-th roots: $w_{n,k} = (w_n)^k$.

The **discrete Fourier transform (DFT)** of the polynomial $A(x)$ (or equivalently the vector of coefficients $(a_0, a_1, \ldots, a_{n-1})$ is defined as the values of the polynomial at the points $x = w_{n,k}$, i.e. it is the vector:

$$\begin{aligned}\text{DFT}(a_0, a_1, \ldots, a_{n-1}) &= (y_0, y_1, \ldots, y_{n-1}) \\ &= (A(w_{n_0}), A(w_{n,1}), \ldots, A(w_{n,n-1} \\ &= (A(w_n^0), A(w_n^1), \ldots, A(w_n^{n-1}))\end{aligned}$$

Similarly the **inverse discrete Fourier transform** is defined: The inverse DFT of values of the polynomial $(y_0, y_1, \ldots, y_{n-1})$ are the coefficients of the polynomial $(a_0, a_1, \ldots, a_{n-1})$.

$$\text{InverseDFT}(y_0, y_1, \ldots, y_{n-1}) = (a_0, a_1, \ldots, a_{n-1})$$

Thus, if a direct DFT computes the values of the polynomial at the points at the n-th roots, the inverse

DFT can restore the coefficients of the polynomial using those values.

Application of the DFT: fast multiplication of polynomials

Let there be two polynomials A and B. We compute the DFT for each of them: $\mathrm{DFT}(A)$ and $\mathrm{DFT}(B)$.

What happens if we multiply these polynomials? Obviously at each point the values are simply multiplied, i.e.

$$(A \cdot B)(x) = A(x) \cdot B(x).$$

This means that if we multiply the vectors $\mathrm{DFT}(A)$ and $\mathrm{DFT}(B)$ - by multiplying each element of one vector by the corresponding element of the other vector - then we get nothing other than the DFT of the polynomial $\mathrm{DFT}(A \cdot B)$:

$$\mathrm{DFT}(A \cdot B) = \mathrm{DFT}(A) \cdot \mathrm{DFT}(B)$$

Finally, applying the inverse DFT, we obtain:

$$A \cdot B = \mathrm{InverseDFT}(\mathrm{DFT}(A) \cdot \mathrm{DFT}(B))$$

On the right the product of the two DFTs we mean the pairwise product of the vector elements. This can be computed in $O(n)$ time. If we can compute the DFT and the inverse DFT in $O(n \log n)$, then we can compute the product of the two polynomials (and consequently also two long numbers) with the same time complexity.

It should be noted, that the two polynomials should have the same degree. Otherwise the two result vectors of the DFT have different length. We can accomplish this by adding coefficients with the value 0.

And also, since the result of the product of two polynomials is a polynomial of degree $2(n-1)$, we have to double the degrees of each polynomial (again by padding 0s). From a vector with n values we cannot reconstruct the desired polynomial with $2n-1$ coefficients.

Fast Fourier Transform

The **fast Fourier transform** is a method that allows computing the DFT in $O(n \log n)$ time. The basic idea of the FFT is to apply divide and conquer. We divide the coefficient vector of the polynomial into two vectors, recursively compute the DFT for each of them, and combine the results to compute the DFT of the complete polynomial.

So let there be a polynomial $A(x)$ with degree $n-1$, where n is a power of 2, and $n > 1$:

$$A(x) = a_0 x^0 + a_1 x^1 + \cdots + a_{n-1} x^{n-1}$$

We divide it into two smaller polynomials, the one containing only the coefficients of the even positions, and the one containing the coefficients of the odd positions:

$$\begin{aligned} A_0(x) &= a_0 x^0 + a_2 x^1 + \cdots + a_{n-2} x^{\frac{n}{2}-1} \\ A_1(x) &= a_1 x^0 + a_3 x^1 + \cdots + a_{n-1} x^{\frac{n}{2}-1} \end{aligned}$$

It is easy to see that

$$A(x) = A_0(x^2) + x A_1(x^2).$$

The polynomials A_0 and A_1 are only half as much coefficients as the polynomial A. If we can compute the $\mathrm{DFT}(A)$ in linear time using $\mathrm{DFT}(A_0)$ and $\mathrm{DFT}(A_1)$, then we get the recurrence $T_{\mathrm{DFT}}(n) = 2T_{\mathrm{DFT}}\left(\frac{n}{2}\right) + O(n)$ for the time

complexity, which results in $T_{\text{DFT}}(n) = O(n \log n)$ by the **master theorem**.

Let's learn how we can accomplish that.

Suppose we have computed the vectors $\left(y_k^0\right)_{k=0}^{n/2-1} = \text{DFT}(A_0)$ and $\left(y_k^1\right)_{k=0}^{n/2-1} = \text{DFT}(A_1)$. Let us find a expression for $(y_k)_{k=0}^{n-1} = \text{DFT}(A)$.

For the first $\frac{n}{2}$ values we can just use the previously noted equation $A(x) = A_0(x^2) + xA_1(x^2)$:

$$y_k = y_k^0 + w_n^k y_k^1, \quad k = 0 \dots \frac{n}{2} - 1.$$

However for the second $\frac{n}{2}$ values we need to find a slightly, different expression:

$$\begin{aligned}
y_{k+n/2} &= A\left(w_n^{k+n/2}\right) \\
&= A_0\left(w_n^{2k+n}\right) + w_n^{k+n/2} A_1\left(w_n^{2k+n}\right) \\
&= A_0\left(w_n^{2k} w_n^n\right) + w_n^k w_n^{n/2} A_1\left(w_n^{2k} w_n^n\right) \\
&= A_0\left(w_n^{2k}\right) - w_n^k A_1\left(w_n^{2k}\right) \\
&= y_k^0 - w_n^k y_k^1
\end{aligned}$$

Here we used again $A(x) = A_0(x^2) + xA_1(x^2)$ and the two identities $w_n^n = 1$ and $w_n^{n/2} = -1$.

Therefore we get the desired formulas for computing the whole vector (y_k):

$$\begin{aligned}
y_k &= y_k^0 + w_n^k y_k^1, \qquad && k = 0 \dots \frac{n}{2} - 1, \\
y_{k+n/2} &= y_k^0 - w_n^k y_k^1, \qquad && k = 0 \dots \frac{n}{2} - 1.
\end{aligned}$$

(This pattern $a + b$ and $a - b$ is sometimes called a **butterfly**.)

Thus we learned how to compute the DFT in $O(n \log n)$ time.

Inverse FFT

Let the vector $(y_0, y_1, \dots y_{n-1})$ - the values of polynomial A of degree $n-1$ in the points $x = w_n^k$ - be given. We want to restore the coefficients $(a_0, a_1, \dots, a_{n-1})$ of the polynomial. This known problem is called **interpolation**, and there are general algorithms for solving it. But in this special case (since we know the values of the points at the roots of unity), we can obtains a much simpler algorithm (that is practically the same as the direct FFT).

We can write the DFT, according to its definition, in the matrix form:

$$\begin{pmatrix} w_n^0 & w_n^0 & w_n^0 & w_n^0 & \cdots & w_n^0 \\ w_n^0 & w_n^1 & w_n^2 & w_n^3 & \cdots & w_n^{n-1} \\ w_n^0 & w_n^2 & w_n^4 & w_n^6 & \cdots & w_n^{2(n-1)} \\ w_n^0 & w_n^3 & w_n^6 & w_n^9 & \cdots & w_n^{3(n-1)} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ w_n^0 & w_n^{n-1} & w_n^{2(n-1)} & w_n^{3(n-1)} & \cdots & w_n^{(n-1)(n-1)} \end{pmatrix} \begin{pmatrix} a$$

This matrix is called the **Vandermonde matrix**.

Thus we can compute the vector $(a_0, a_1, \dots, a_{n-1})$ by multiplying the vector $(y_0, y_1, \dots y_{n-1})$ from the left with the inverse of the matrix:

$$\begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ \vdots \\ a_{n-1} \end{pmatrix} = \begin{pmatrix} w_n^0 & w_n^0 & w_n^0 & w_n^0 & \cdots & w \\ w_n^0 & w_n^1 & w_n^2 & w_n^3 & \cdots & w \\ w_n^0 & w_n^2 & w_n^4 & w_n^6 & \cdots & w_n^2 \\ w_n^0 & w_n^3 & w_n^6 & w_n^9 & \cdots & w_n^3 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \\ w_n^0 & w_n^{n-1} & w_n^{2(n-1)} & w_n^{3(n-1)} & \cdots & w_n^{(n-} \end{pmatrix}$$

A quick check can verify that the inverse of the matrix has the following form:

$$\frac{1}{n} \begin{pmatrix} w_n^0 & w_n^0 & w_n^0 & w_n^0 & \cdots & w_n^0 \\ w_n^0 & w_n^{-1} & w_n^{-2} & w_n^{-3} & \cdots & w_n^{-(n-1)} \\ w_n^0 & w_n^{-2} & w_n^{-4} & w_n^{-6} & \cdots & w_n^{-2(n-1)} \\ w_n^0 & w_n^{-3} & w_n^{-6} & w_n^{-9} & \cdots & w_n^{-3(n-1)} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ w_n^0 & w_n^{-(n-1)} & w_n^{-2(n-1)} & w_n^{-3(n-1)} & \cdots & w_n^{-(n-1)(n-} \end{pmatrix}$$

Thus we obtain the formula:

$$a_k = \frac{1}{n} \sum_{j=0}^{n-1} y_j w_n^{-kj}$$

Comparing this to the formula for y_k

$$y_k = \sum_{j=0}^{n-1} a_j w_n^{kj},$$

we notice that these problems are almost the same, so the coefficients a_k can be found by the same divide and conquer algorithm, as well as the direct FFT, only instead of w_n^k we have to use w_n^{-k}, and at the end we need to divide the resulting coefficients by n.

Thus the computation of the inverse DFT is almost the same as the calculation of the direct DFT, and it also can be performed in $O(n \log n)$ time.

Implementation

Here we present a simple recursive **implementation of the FFT** and the inverse FFT, both in one function, since the difference between the forward and the inverse FFT are so minimal. To store the complex numbers we use the complex type in the C++ STL.

```
using cd = complex<double>;
const double PI = acos(-1);

void fft(vector<cd> & a, bool invert) {
    int n = a.size();
    if (n == 1)
        return;

    vector<cd> a0(n / 2), a1(n / 2);
    for (int i = 0; 2 * i < n; i++) {
        a0[i] = a[2*i];
        a1[i] = a[2*i+1];
    }
    fft(a0, invert);
    fft(a1, invert);

    double ang = 2 * PI / n * (invert ? -1 : 1
    cd w(1), wn(cos(ang), sin(ang));
    for (int i = 0; 2 * i < n; i++) {
        a[i] = a0[i] + w * a1[i];
        a[i + n/2] = a0[i] - w * a1[i];
        if (invert) {
            a[i] /= 2;
            a[i + n/2] /= 2;
        }
        w *= wn;
```

```
    }
}
```

The function gets passed a vector of coefficients, and the function will compute the DFT or inverse DFT and store the result again in this vector. The argument invert shows whether the direct or the inverse DFT should be computed. Inside the function we first check if the length of the vector is equal to one, if this is the case then we don't have to do anything. Otherwise we divide the vector a into two vectors $a0$ and $a1$ and compute the DFT for both recursively. Then we initialize the value wn and a variable w, which will contain the current power of wn. Then the values of the resulting DFT are computed using the above formulas.

If the flag invert is set, then we replace wn with wn^{-1}, and each of the values of the result is divided by 2 (since this will be done in each level of the recursion, this will end up dividing the final values by n).

Using this function we can create a function for **multiplying two polynomials**:

```
vector<int> multiply(vector<int> const& a, vec
    vector<cd> fa(a.begin(), a.end()), fb(b.be
    int n = 1;
    while (n < a.size() + b.size())
        n <<= 1;
    fa.resize(n);
    fb.resize(n);

    fft(fa, false);
    fft(fb, false);
    for (int i = 0; i < n; i++)
        fa[i] *= fb[i];
    fft(fa, true);
```

```
    vector<int> result(n);
    for (int i = 0; i < n; i++)
        result[i] = round(fa[i].real());
    return result;
}
```

This function works with polynomials with integer coefficients, however you can also adjust it to work with other types. Since there is some error when working with complex numbers, we need round the resulting coefficients at the end.

Finally the function for **multiplying** two long numbers practically doesn't differ from the function for multiplying polynomials. The only thing we have to do afterwards, is to normalize the number:

```
    int carry = 0;
    for (int i = 0; i < n; i++)
        result[i] += carry;
        carry = result[i] / 10;
        result[i] %= 10;
    }
```

Since the length of the product of two numbers never exceed the total length of both numbers, the size of the vector is enough to perform all carry operations.

Improved implementation: in-place computation

To increase the efficiency we will switch from the recursive implementation to an iterative one. In the above recursive implementation we explicitly separated the vector a into two vectors - the element on the even positions got assigned to one temporary vector, and the elements on odd positions to another. However if we reorder the elements in a certain way, we don't need to

create these temporary vectors (i.e. all the calculations can be done "in-place", right in the vector A itself).

Note that at the first recursion level, the elements whose lowest bit of the position was zero got assigned to the vector a_0, and the ones with a one as the lowest bit of the position got assigned to a_1. In the second recursion level the same thing happens, but with the second lowest bit instead, etc. Therefore if we reverse the bits of the position of each coefficient, and sort them by these reversed values, we get the desired order (it is called the bit-reversal permutation).

For example the desired order for $n = 8$ has the form:

$$a = \{[(a_0, a_4), (a_2, a_6)], [(a_1, a_5), (a_3, a_7)]\}$$

Indeed in the first recursion level (surrounded by curly braces), the vector gets divided into two parts $[a_0, a_2, a_4, a_6]$ and $[a_1, a_3, a_5, a_7]$. As we see, in the bit-reversal permutation this corresponds to simply dividing the vector into two halves: the first $\frac{n}{2}$ elements and the last $\frac{n}{2}$ elements. Then there is a recursive call for each halve. Let the resulting DFT for each of them be returned in place of the elements themselves (i.e. the first half and the second half of the vector a respectively.

$$a = \{[y_0^0, y_1^0, y_2^0, y_3^0], [y_0^1, y_1^1, y_2^1, y_3^1]\}$$

Now we want to combine the two DFTs into one for the complete vector. The order of the elements is ideal, and we can also perform the union directly in this vector. We can take the elements y_0^0 and y_0^1 and perform the butterfly transform. The place of the resulting two values is the same as the place of the two initial values, so we get:

$$a = \{[y_0^0 + w_n^0 y_0^1, y_1^0, y_2^0, y_3^0], [y_0^0 - w_n^0 y_0^1, y_1^1, y_2^1, y_3^1]\}$$

Similarly we can compute the butterfly transform of y_1^0 and y_1^1 and put the results in their place, and so on. As a result we get:

$$a = \{[y_0^0 + w_n^0 y_0^1, y_1^0 + w_n^1 y_1^1, y_2^0 + w_n^2 y_2^1, y_3^0 + w_n^3 y_3^1], [y_0^0 -$$

Thus we computed the required DFT from the vector a.

Here we described the process of computing the DFT only at the first recursion level, but the same works obviously also for all other levels. Thus, after applying the bit-reversal permutation, we can compute the DFT in-place, without any additional memory.

This additionally allows us to get rid of the recursion. We just start at the lowest level, i.e. we divide the vector into pairs and apply the butterfly transform to them. This results with the vector a with the work of the last level applied. In the next step we divide the vector into vectors of size 4, and again apply the butterfly transform, which gives us the DFT for each block of size 4. And so on. Finally in the last step we obtained the result of the DFTs of both halves of a, and by applying the butterfly transform we obtain the DFT for the complete vector a.

```
using cd = complex<double>;
const double PI = acos(-1);

int reverse(int num, int lg_n) {
    int res = 0;
    for (int i = 0; i < lg_n; i++) {
        if (num & (1 << i))
            res |= 1 << (lg_n - 1 - i);
    }
    return res;
}

void fft(vector<cd> & a, bool invert) {
    int n = a.size();
```

```
    int lg_n = 0;
    while ((1 << lg_n) < n)
        lg_n++;

    for (int i = 0; i < n; i++) {
        if (i < reverse(i, lg_n))
            swap(a[i], a[reverse(i, lg_n)]);
    }

    for (int len = 2; len <= n; len <<= 1) {
        double ang = 2 * PI / len * (invert ?
        cd wlen(cos(ang), sin(ang));
        for (int i = 0; i < n; i += len) {
            cd w(1);
            for (int j = 0; j < len / 2; j++)
                cd u = a[i+j], v = a[i+j+len/2
                a[i+j] = u + v;
                a[i+j+len/2] = u - v;
                w *= wlen;
            }
        }
    }

    if (invert) {
        for (cd & x : a)
            x /= n;
    }
}
```

At first we apply the bit-reversal permutation by swapping the each element with the element of the reversed position. Then the $\log n - 1$ states of the algorithm we compute the DFT for each block of the corresponding size len. For all those blocks we have the same root of unity wlen. We iterate all blocks and perform the butterfly transform on each of them.

We can further optimize the reversal of the bits. In the previous implementation we iterated all bits of the index

and created the bitwise reversed index. However we can reverse the bits in a different way.

Suppose that j already contains the reverse of i. Then by to go to $i+1$, we have to increment i, and we also have to increment j, but in a "reversed" number system. Adding one in the conventional binary system is equivalent to flip all tailing ones into zeros and flipping the zero right before them into a one. Equivalently in the "reversed" number system, we flip all leading ones, and the also the next zero.

Thus we get the following implementation:

```
using cd = complex<double>;
const double PI = acos(-1);

void fft(vector<cd> & a, bool invert) {
    int n = a.size();

    for (int i = 1, j = 0; i < n; i++) {
        int bit = n >> 1;
        for (; j & bit; bit >>= 1)
            j ^= bit;
        j ^= bit;

        if (i < j)
            swap(a[i], a[j]);
    }

    for (int len = 2; len <= n; len <<= 1) {
        double ang = 2 * PI / len * (invert ?
        cd wlen(cos(ang), sin(ang));
        for (int i = 0; i < n; i += len) {
            cd w(1);
            for (int j = 0; j < len / 2; j++)
                cd u = a[i+j], v = a[i+j+len/2
                a[i+j] = u + v;
                a[i+j+len/2] = u - v;
```

```
                w *= wlen;
            }
        }
    }

    if (invert) {
        for (cd & x : a)
            x /= n;
    }
}
```

Additionally we can precompute the bit-reversal permutation beforehand. This is especially useful when the size n is the same for all calls. But even when we only have three calls (which are necessary for multiplying two polynomials), the effect is noticeable. Also we can precompute all roots of unity and their powers.

Number theoretic transform

Now we switch the objective a little bit. We still want to multiply two polynomials in $O(n \log n)$ time, but this time we want to compute the coefficients modulo some prime number p. Of course for this task we can use the normal DFT and apply the modulo operator to the result. However, doing so might lead to rounding errors, especially when dealing with large numbers. The **number theoretic transform (NTT)** has the advantage, that it only works with integer, and therefore the result are guaranteed to be correct.

The discrete Fourier transform is based on complex numbers, and the n-th roots of unity. To efficiently compute it, we extensively use properties of the roots (e.g. that there is one root that generates all other roots by exponentiation).

But the same properties hold for the n-th roots of unity in modular arithmetic. A n-th root of unity under a primitive field is such a number w_n that satisfies:

$$\begin{aligned}(w_n)^n &= 1 \pmod p,\\ (w_n)^k &\neq 1 \pmod p, \quad 1 \leq k < n.\end{aligned}$$

The other $n-1$ roots can be obtained as powers of the root w_n.

To apply it in the fast Fourier transform algorithm, we need a root to exist for some n, which is a power of 2, and also for all smaller powers. We can notice the following interesting property:

$$\begin{aligned}(w_n^2)^m = w_n^n &= 1 \pmod p, \quad \text{with } m = \frac{n}{2}\\ (w_n^2)^k = w_n^{2k} &\neq 1 \pmod p, \quad 1 \leq k < m.\end{aligned}$$

Thus if w_n is a n-th root of unity, then w_n^2 is a $\frac{n}{2}$-th root of unity. And consequently for all smaller powers of two there exist roots of the required degree, and they can be computed using w_n.

For computing the inverse DFT, we need the inverse w_n^{-1} of w_n. But for a prime modulus the inverse always exists.

Thus all the properties that we need from the complex roots are also available in modular arithmetic, provided that we have a large enough module p for which a n-th root of unity exists.

For example we can take the following values: module $p = 7340033$, $w_{2^{20}} = 5$. If this module is not enough, we need to find a different pair. We can use that fact that for modules of the form $p = c2^k + 1$ (and p is prime), there always exists the 2^k-th root of unity. It can be

shown that g^c is such a 2^k-th root of unity, where g is a primitive root of p.

```
const int mod = 7340033;
const int root = 5;
const int root_1 = 4404020;
const int root_pw = 1 << 20;

void fft(vector<int> & a, bool invert) {
    int n = a.size();

    for (int i = 1, j = 0; i < n; i++) {
        int bit = n >> 1;
        for (; j & bit; bit >>= 1)
            j ^= bit;
        j ^= bit;

        if (i < j)
            swap(a[i], a[j]);
    }

    for (int len = 2; len <= n; len <<= 1) {
        int wlen = invert ? root_1 : root;
        for (int i = len; i < root_pw; i <<= 1
            wlen = (int)(1LL * wlen * wlen % m

        for (int i = 0; i < n; i += len) {
            int w = 1;
            for (int j = 0; j < len / 2; j++)
                int u = a[i+j], v = (int)(1LL
                a[i+j] = u + v < mod ? u + v :
                a[i+j+len/2] = u - v >= 0 ? u
                w = (int)(1LL * w * wlen % mod
            }
        }
    }

    if (invert) {
        int n_1 = inverse(n, mod);
```

```
            for (int & x : a)
                x = (int)(1LL * x * n_1 % mod);
        }
    }
```

Here the function `inverse` computes the modular inverse (see Modular Multiplicative Inverse). The constants `mod`, `root`, `root_pw` determine the module and the root, and `root_1` is the inverse of `root` modulo `mod`.

In practice this implementation is slower than the implementation using complex numbers (due to the huge number of modulo operations), but it has some advantages such as less memory usage and no rounding errors.

Multiplication with arbitrary modulus

Here we want to achieve the same goal as in previous section. Multiplying two polynomial $A(x)$ and $B(x)$, and computing the coefficients modulo some number M. The number theoretic transform only works for certain prime numbers. What about the case when the modulus is not of the desired form?

One option would be to perform multiple number theoretic transforms with different prime numbers of the form $c2^k + 1$, then apply the Chinese Remainder Theorem to compute the final coefficients.

Another options is to distribute the polynomials $A(x)$ and $B(x)$ into two smaller polynomials each

$$A(x) = A_1(x) + A_2(x) \cdot C$$
$$B(x) = B_1(x) + B_2(x) \cdot C$$

with $C \approx \sqrt{M}$.

Then the product of $A(x)$ and $B(x)$ can then be represented as:

$$A(x) \cdot B(x) = A_1(x) \cdot B_1(x) + (A_1(x) \cdot B_2(x) + A_2(x) \cdot$$

The polynomials $A_1(x)$, $A_2(x)$, $B_1(x)$ and $B_2(x)$ contain only coefficients smaller than $\sqrt{M}$, therefore the coefficients of all the appearing products are smaller than $M \cdot n$, which is usually small enough to handle with typical floating point types.

This approach therefore requires computing the products of polynomials with smaller coefficients (by using the normal FFT and inverse FFT), and then the original product can be restored using modular addition and multiplication in $O(n)$ time.

Applications

DFT can be used in a huge variety of other problems, which at the first glance have nothing to do with multiplying polynomials.

All possible sums

We are given two arrays $a[]$ and $b[]$. We have to find all possible sums $a[i] + b[j]$, and for each sum count how often it appears.

For example for $a = [1,\ 2,\ 3]$ and $b = [2,\ 4]$ we get: then sum 3 can be obtained in 1 way, the sum 4 also in 1 way, 5 in 2, 6 in 1, 7 in 1.

We construct for the arrays a and b two polynomials A and B. The numbers of the array will act as the exponents in the polynomial ($a[i] \Rightarrow x^{a[i]}$); and the coefficients of this term will by how often the number appears in the array.

Then, by multiplying these two polynomials in $O(n \log n)$ time, we get a polynomial C, where the exponents will tell us which sums can be obtained, and the coefficients tell us how often. To demonstrate this on the example:

$$(1x^1 + 1x^2 + 1x^3)(1x^2 + 1x^4) = 1x^3 + 1x^4 + 2x^5 + 1x^6 +$$

All possible scalar products

We are given two arrays $a[]$ and $b[]$ of length n. We have to compute the products of a with every cyclic shift of b.

We generate two new arrays of size $2n$: We reverse a and append n zeros to it. And we just append b to itself. When we multiply these two arrays as polynomials, and look at the coefficient $c[n-1],\ c[n],\ c[2n-2]$ of the product c, we get:

$$c[k] = \sum_{i+j=k} a[i]b[j]$$

And since all the elements $a[i] = 0$ for $i \geq n$:

$$c[k] = \sum_{i=0}^{n-1} a[i]b[k-i]$$

It is easy to see that this sum is just the scalar product of the vector a with the $(k-(n-1))$-th cyclic left shift of b. Thus these coefficients are the answer to the problem, and we were still able to obtain it in $O(n \log n)$ time. Note here that $c[2n-1]$ also gives us the n-th cyclic shift but that is the same as the 0-th cyclic shift so we don't need to consider that separately into our answer.

Two stripes

We are given two Boolean stripes (cyclic arrays of values 0 and 1) a and b. We want to find all ways to attach the first stripe to the second one, such that at no position we have a 1 of the first stripe next to a 1 of the second stripe.

The problem doesn't actually differ much from the previous problem. Attaching two stripes just means that we perform a cyclic shift on the second array, and we can attach the two stripes, if scalar product of the two arrays is 0.

String matching

We are given two strings, a text T and a pattern P, consisting of lowercase letters. We have to compute all the occurrences of the pattern in the text.

We create a polynomial for each string ($T[i]$ and $P[I]$ are numbers between 0 and 25 corresponding to the 26 letters of the alphabet):

$$A(x) = a_0 x^0 + a_1 x^1 + \cdots + a_{n-1} x^{n-1}, \quad n = |T|$$

with

$$a_i = \cos(\alpha_i) + i \sin(\alpha_i), \quad \alpha_i = \frac{2\pi T[i]}{26}.$$

And

$$B(x) = b_0 x^0 + b_1 x^1 + \cdots + b_{m-1} x^{m-1}, \quad m = |P|$$

with

$$b_i = \cos(\beta_i) - i \sin(\beta_i), \quad \beta_i = \frac{2\pi P[m-i-1]}{26}.$$

Notice that with the expression $P[m-i-1]$ explicitly reverses the pattern.

The $(m-1+i)$th coefficients of the product of the two polynomials $C(x) = A(x) \cdot B(x)$ will tell us, if the pattern appears in the text at position i.

$$c_{m-1+i} = \sum_{j=0}^{m-1} a_{i+j} \cdot b_{m-1-j} = \sum_{j=0}^{m-1} (\cos(\alpha_{i+j}) + i \sin(\alpha_{i+j}))$$

with $\alpha_{i+j} = \frac{2\pi T[i+j]}{26}$ and $\beta_j = \frac{2\pi P[j]}{26}$

If there is a match, than $T[i+j] = P[j]$, and therefore $\alpha_{i+j} = \beta_j$. This gives (using the Pythagorean trigonometric identity):

$$\begin{aligned} c_{m-1+i} &= \sum_{j=0}^{m-1} (\cos(\alpha_{i+j}) + i \sin(\alpha_{i+j})) \cdot (\cos(\alpha_{i+j}) - i \sin \\ &= \sum_{j=0}^{m-1} \cos(\alpha_{i+j})^2 + \sin(\alpha_{i+j})^2 = \sum_{j=0}^{m-1} 1 = m \end{aligned}$$

If there isn't a match, then at least a character is different, which leads that one of the products $a_{i+1} \cdot b_{m-1-j}$ is not equal to 1, which leads to the coefficient $c_{m-1+i} \neq m$.

String matching with wildcards

This is an extension of the previous problem. This time we allow that the pattern contains the wildcard character $*$, which can match every possible letter. E.g. the pattern $a*c$ appears in the text $abccaacc$ at exactly three positions, at index 0, index 4 and index 5.

We create the exact same polynomials, except that we set $b_i = 0$ if $P[m-i-1] = *$. If x is the number of wildcards in P, then we will have a match of P in T at index i if $c_{m-1+i} = m - x$.

Practice problems

- SPOJ - POLYMUL
- SPOJ - MAXMATCH
- SPOJ - ADAMATCH
- Codeforces - Yet Another String Matching Problem
- Codeforces - Lightsabers (hard)
- Kattis - K-Inversions
- Codeforces - Dasha and cyclic table