CP-Algorithms

Page Authors

Search

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:

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

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:

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

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 $T_{\mathrm{DFT}}(n) = O(n \log n)$ by the **master theorem**.

Let's learn how we can accomplish that.

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

For the first $\frac{n}{2}$ values we can just use the previously noted equation $A(x) = A_0(x^2) + x A_1(x^2)$:

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

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:

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

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:

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

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

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

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: