Fast Fourier Transform for the O (N log N). Application to the multiplication of two polynomials or long numbers

Here we consider an algorithm which allows to multiply two polynomials of length n during $O(n \log n)$ that time much better $O(n^2)$ achievable trivial multiplication algorithm. Obviously, the multiplication of two long numbers can be reduced to the multiplication of polynomials, so the two long numbers can also multiply during $O(n \log n)$.

Fast Fourier Transform invention is attributed to Cooley (Coolet) and Taki (Tukey) - 1965 actually invented a FFT repeatedly before, but its importance is not fully recognized until the advent of modern computers. Some researchers attribute the discovery FFT Runge (Runge) and Koenig (Konig) Finally in 1924, the discovery of this method is attributed to more Gaussian (Gauss) in 1805

Discrete Fourier transform (DFT)

Suppose there is a polynomial of n the second degree:

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

Without loss of generality, we can assume that n is a power of 2. If in fact n not a power of 2, then we just add the missing coefficients, setting them equal to zero.

Theory of functions of a complex variable is known that complex roots n-th degree of unity exists exactly n. We denote these roots by $w_{n,k}, k = 0 \ldots n-1$ then known that $w_{n,k} = e^{i\frac{2\pi k}{n}}$. In addition, one of these roots $w_n = w_{n,1} = e^{i\frac{2\pi}{n}}$ (called the principal value of the root n-th degree of unity) is that all the other roots are his powers: $w_{n,k} = (w_n)^k$.

Then **the discrete Fourier transform (DFT)** (discrete Fourier transform, DFT) of the polynomial $A(x)$ (or, equivalently, the DFT vector of coefficients $(a_0, a_1, \ldots, a_{n-1})$) are the values of this polynomial at the points $x = w_{n,k}$, ie is the vector:

$$\mathrm{DFT}(a_0, a_1, \ldots, a_{n-1}) = (y_0, y_1, \ldots, y_{n-1}) = (A(w_{n,0}), A(w_{n,1}), \ldots, A(w_{n,n-1})) =$$

$$= (A(w_n^0), A(w_n^1), \ldots, A(w_n^{n-1})).$$

Similarly defined and **inverse discrete Fourier transform** (InverseDFT). Inverse DFT to the vector of a polynomial $(y_0, y_1, \ldots y_{n-1})$- is the vector of 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 the direct proceeds from DFT coefficients of the polynomial to its values in the complex roots of $n$the second degree of unity, then the inverse DFT - on the contrary, by the values of the coefficients of the polynomial recovers.

Application of DFT for fast multiplication of polynomials

Given two polynomials $A$and B. Calculate the DFT for each of them: $\text{DFT}(A)$and $\text{DFT}(B)$- two vector values of the polynomials.

Now, what happens when the multiplication of polynomials? Obviously, in each point of their values are simply multiplied, that is,

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

But this means that if we multiply the vector $\text{DFT}(A)$and $\text{DFT}(B)$, by simply multiplying each element of a vector to the corresponding element of another vector, then we get nothing else than the DFT of a polynomial $A \times B$:

$$\text{DFT}(A \times B) = \text{DFT}(A) \times \text{DFT}(B).$$

Finally, applying the inverse DFT, we obtain:

$$A \times B = \text{InverseDFT}(\text{DFT}(A) \times \text{DFT}(B)),$$

where, again, right under the product of two DFT mean pairwise products of the elements of the vectors. This work obviously requires to compute only $O(n)$operations. Thus, if we learn how to calculate the DFT and inverse DFT in time $O(n \log n)$, the product of two polynomials (and, consequently, the two long numbers) we can find for the same asymptotic behavior.

It should be noted that the first two polynomials to be lead to the same degree (simply adding one of these coefficients with zeros.) Second, the product of two polynomials of degree $n$polynomial of degree obtained $2n - 1$, so that the result is correct, you need to double the pre-degree of each polynomial (again, adding to their zero coefficients).

Fast Fourier transform

Fast Fourier transform (fast Fourier transform) - a method to calculate the DFT for time $O(n \log n)$. This method relies on the properties of the complex roots of unity (namely, that some degree of give other roots roots).

The basic idea is to divide the FFT coefficient vector into two vectors, the recursive computation of the DFT for them and merging the results into a single FFT.

Thus, suppose there is a polynomial $A(x)$of degree n, where n- a power of two, and $n > 1$:

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

Divide it into two polynomials, one - with even and the other - with the odd coefficients:

$$A_0(x) = a_0 x^0 + a_2 x^1 + \ldots + a_{n-2} x^{n/2-1},$$

$$A_1(x) = a_1 x^0 + a_3 x^1 + \ldots + a_{n-1} x^{n/2-1}.$$

It is easy to verify that:

$$A(x) = A_0(x^2) + x A_1(x^2). \qquad (1)$$

The polynomials $A_0$and A_1have twice lower degree than the polynomial A. If we can in linear time from the calculated $\mathrm{DFT}(A_0)$and $\mathrm{DFT}(A_1)$calculate $\mathrm{DFT}(A)$, and then we get the desired fast Fourier transform algorithm (since it is a standard chart of "divide and conquer", and it is known asymptotic estimate $O(n \log n)$).

So, suppose we have calculated the vector $\{y_k^0\}_{k=0}^{n/2-1} = \mathrm{DFT}(A_0)$and $\{y_k^1\}_{k=0}^{n/2-1} = \mathrm{DFT}(A_1)$. Find expressions for $\{y_k\}_{k=0}^{n-1} = \mathrm{DFT}(A)$.

First, recalling (1), we immediately obtain the values for the first half of the coefficients:

$$y_k = y_k^0 + w_n^k y_k^1, \quad k = 0 \ldots n/2 - 1.$$

For the second half of the coefficients after transformation also get a simple formula:

$$y_{k+n/2} = A(w_n^{k+n/2}) = A_0(w_n^{2k+n}) + w_n^{k+n/2} A_1(w_n^{2k+n}) = A_0(w_n^{2k} w_n^n) + w_n^k w_n^{n/2} A_1(w_n^{2k} w_n^n) =$$

$$= A_0(w_n^{2k}) - w_n^k A_1(w_n^{2k}) = y_k^0 - w_n^k y_k^1.$$

(Here we have used (1), as well as identities $w_n^n = 1$, $w_n^{n/2} = -1$.)

So as a result we got the formula for calculating the total vector $\{y_k\}$:

$$y_k = y_k^0 + w_n^k y_k^1, \quad k = 0 \dots n/2 - 1,$$

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

(These formulas, ie two formulas of the form $a + bc$and $a - bc$are sometimes called "butterfly transformation" ("butterfly operation"))

Thus, we finally built the FFT algorithm.

IFFT

So, let a vector $(y_0, y_1, \dots, y_{n-1})$- values of a polynomial $A$of degree $n$at points $x = w_n^k$. Need to recover the coefficients $(a_0, a_1, \dots, a_{n-1})$of the polynomial. This well-known problem is called **interpolation** , for this problem there are some common algorithms for solving, but in this case is obtained by a very simple algorithm (a simple fact that it does not differ from the direct FFT).

DFT, we can write, according to his definition, in matrix form:

$$\begin{pmatrix} w_n^0 & w_n^0 & w_n^0 & w_n^0 & \cdots & w_n^0 \\ w_n^0 & w_n^1 & w_n^2 & w_n^3 & \cdots & w_n^{n-1} \\ w_n^0 & w_n^2 & w_n^4 & w_n^6 & \cdots & w_n^{2(n-1)} \\ w_n^0 & w_n^3 & w_n^6 & w_n^9 & \cdots & w_n^{3(n-1)} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ w_n^0 & w_n^{n-1} & w_n^{2(n-1)} & w_n^{3(n-1)} & \cdots & w_n^{(n-1)(n-1)} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ \vdots \\ a_{n-1} \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_{n-1} \end{pmatrix}.$$

The vector $(a_0, a_1, \dots, a_{n-1})$can be found by multiplying the vector $(y_0, y_1, \dots, y_{n-1})$by the inverse matrix to the matrix, stands to the left (which, incidentally, is called Vandermonde matrix):

$$\begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ \vdots \\ a_{n-1} \end{pmatrix} = \begin{pmatrix} w_n^0 & w_n^0 & w_n^0 & w_n^0 & \cdots & w_n^0 \\ w_n^0 & w_n^1 & w_n^2 & w_n^3 & \cdots & w_n^{n-1} \\ w_n^0 & w_n^2 & w_n^4 & w_n^6 & \cdots & w_n^{2(n-1)} \\ w_n^0 & w_n^3 & w_n^6 & w_n^9 & \cdots & w_n^{3(n-1)} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ w_n^0 & w_n^{n-1} & w_n^{2(n-1)} & w_n^{3(n-1)} & \cdots & w_n^{(n-1)(n-1)} \end{pmatrix}^{-1} \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_{n-1} \end{pmatrix}.$$

Direct verification shows that this inverse matrix is as follows:

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

Thus, we obtain:

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

Comparing it with the formula for y_k:

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

we notice that these two problems are not real, so that the coefficients $a_k$can be found in the same algorithm "divide and rule" as a direct FFT, but instead w_n^kshould be used everywhere w_n^{-k}, and every element of the result should be divided into n.

Thus, the calculation of the inverse DFT is not very different from the direct computation of the DFT, and can also be performed during $O(n \log n)$.

Implementation

Consider the following simple recursive **implementation of the FFT** and inverse FFT to implement them as a single function since the differences between the direct and inverse FFT low. For storing complex numbers to use the standard C + + STL type complex (defined in the header file <complex>).

```
typedef complex<double> base;

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

	vector<base> a0 (n/2),  a1 (n/2);
	for (int i=0, j=0; i<n; i+=2, ++j) {
```

```
            a0[j] = a[i];
            a1[j] = a[i+1];
        }
        fft (a0, invert);
        fft (a1, invert);

        double ang = 2*PI/n * (invert ? -1 : 1);
        base w (1),  wn (cos(ang), sin(ang));
        for (int i=0; i<n/2; ++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;
        }
}
```

In the argument a passed to the function input vector of coefficients, and it will contain the same result. Argument invert shows direct or inverse DFT should be calculated. Inside the function first checks if the length of the vector a is unity, there is nothing else to do - he is the answer. Otherwise, the vector a is split into two vectors $a0$ and $a1$ for which recursively calculated DFT. Then calculated value w_n, and the plant variable w containing the current level w_n. Then calculated by DFT elements resulting formulas described above.

If the flag is specified $\text{invert} = \text{true}$, it w_n is replaced by w_n^{-1}, and each element of the result is divided by 2 (given that these dividing by 2 will take place in each level of recursion, then eventually just turns out that all the elements on the share n).

Then the function for **multiplying two polynomials** is as follows:

```
void multiply (const vector<int> & a, const vector<int> & b,
vector<int> & res) {
    vector<base> fa (a.begin(), a.end()),  fb (b.begin(),
b.end());
    size_t n = 1;
    while (n < max (a.size(), b.size()))  n <<= 1;
    n <<= 1;
    fa.resize (n),  fb.resize (n);

    fft (fa, false),  fft (fb, false);
    for (size_t i=0; i<n; ++i)
```

```
        fa[i] *= fb[i];
    fft (fa, true);

    res.resize (n);
    for (size_t i=0; i<n; ++i)
        res[i] = int (fa[i].real() + 0.5);
}
```

This function works with polynomials with integer coefficients (although, of course, theoretically nothing prevents her work and with fractional coefficients).However, it appears the problem of a large error in the DFT: error can be significant, so round numbers are better most reliable way - by adding 0.5 and then rounding down (**note** : this will not work properly for negative numbers, if any, may appear in your application).

Finally, the function for **multiplying two long numbers** is practically no different from the function for multiplying polynomials. The only feature - that after the multiplication of numbers as they should normalize polynomials, ie perform all the carries bits:

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

(Since the length of the product of two numbers never exceed the total length of numbers, the size of the vector reswill be enough to fulfill all the transfers.)

Improved execution: computing "on the spot" without additional memory

To increase the efficiency abandon recursion explicitly. In the above recursive implementation, we explicitly separated the vector $\mathrm{a}$into two vectors - elements on even positions attributed to the same time creates a vector, and on the odd - to another. However, if we reorder elements in a certain way, the need for creating temporary vectors would then be eliminated (ie, all the calculations we could produce "on the spot" right in the vector a).

Note that the first level of recursion elements, junior (first) position bits are zero, refer to the vector a_0, and the least significant bits positions which are equal to one - to the vector a_1. On the second level of recursion is done the same thing, but for the second bit, etc. So if we are in the position $i$of each element $a[i]$invert the bit order, and reorder elements of the array $a$in accordance with the new indexes, we obtain the desired order

(it is called **bitwise inverse permutation** (bit-reversal permutation)).

For example, in $n = 8$this order is as follows:

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

Indeed, on the first level of recursion (surrounded by curly braces) conventional recursive algorithm is a division of the vector into two parts: $[a_0, a_2, a_4, a_6]$and $[a_1, a_3, a_5, a_7]$. As we can see, bit by bit, this corresponds to the inverse permutation vector simple division into two halves: the first $n/2$element and the last $n/2$element. Then there is a recursive call of each half, and let the resulting DFT from each of them had returned to the place of the elements themselves (ie in the first and second halves of the vector a, respectively):

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

Now we need to merge the two into one DFT for the vector. But the elements stood so well that the union can do right in the array. Indeed, we take the elements $y_0^0$and $y_0^1$is applicable to them transform butterfly, and the result put in their place - and this place and would thereby and which should have been received:

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

Similarly, we apply the transformation to a butterfly $y_1^0$and $y_1^1$the result put in their place, etc. As a result, we obtain:

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

$$\left. \left[y_0^0 - w_n^0 y_0^1,\ y_1^0 - w_n^1 y_1^1,\ y_2^0 - w_n^2 y_2^1,\ y_3^0 - w_n^3 y_3^1 \right] \right\}.$$

Ie We got exactly the desired DFT of the vector a.

We describe the process of calculating the DFT on the first level of recursion, but it is clear that the same arguments are valid for all other levels of recursion.Thus, **after applying the bitwise inverse permutation is possible to calculate the DFT on the spot** , without additional arrays.

But now you can **get rid of the recursion** explicitly. So, we applied bitwise inverse permutation elements. Now do all the work done by the lower level of recursion, ie vector $a$divide into pairs of elements for each applicable conversion butterfly, resulting in the vector $a$will be the results of the lower level of recursion.In the next step the vector divide $a$by four elements, each butterfly transform applied to obtain the result of the DFT for each of the four. And so on, finally, the last step we received the results of the DFT for the two halves of the vector a, it is applicable to the transformation of butterflies and obtain the DFT for the vector a.

Thus, the implementation of:

```
typedef complex<double> base;

int rev (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<base> & a, bool invert) {
      int n = (int) a.size();
      int lg_n = 0;
      while ((1 << lg_n) < n)  ++lg_n;

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

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

```
	}
	if (invert)
		for (int i=0; i<n; ++i)
			a[i] /= n;
}
```

Initially, a vector $a$is used bitwise inverse permutation, for which calculated the number of significant bits ($\lg n$) including n, for each position $i$is the corresponding position, which has a bit write bit representation of the number $i$recorded in the reverse order. If as a result of the resulting position was more i, the elements in these two positions should be exchanged (unless this condition, each pair will exchange twice and in the end nothing happens.)

Then, the $\lg n - 1$algorithm steps, on the $k$second of which ($k = 2 \dots \lg n$) are computed for the DFT block length 2^k. For all of these units will be the same value of a primitive root w_{2^k}, and is stored in a variable wlen. Cycle through $i$iterated by block, and invested in it by the cycle $j$applies the transformation to all elements of the butterfly unit.

You can further **optimize the reverse bits** . In the previous implementation, we explicitly took over all bits of the number, incidentally bitwise inverted order number. However, reverse bits can be performed differently.

For example, suppose that j- already counted the number equal to the inverse permutation of bits number i. Then, during the transition to the next number $i + 1$ we have and the number of $j$add one, but add it to this "inverted" notation. In a conventional binary system to add one - so remove all units standing at the end of the number (ie, younger group of units), and put the unit in front of them. Accordingly, the "inverted" system, we have to go through the bits of the number, starting with the oldest, and while there are ones, delete them and move on to the next bit, and when to meet the first zero bit, put him in a unit and stop.

Thus, we obtain a realization:

```
typedef complex<double> base;

void fft (vector<base> & a, bool invert) {
	int n = (int) 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 ? -1 : 1);
            base wlen (cos(ang), sin(ang));
            for (int i=0; i<n; i+=len) {
                base w (1);
                for (int j=0; j<len/2; ++j) {
                    base u = a[i+j],  v = a[i+j+len/2] * w;
                    a[i+j] = u + v;
                    a[i+j+len/2] = u - v;
                    w *= wlen;
                }
            }
        }
        if (invert)
            for (int i=0; i<n; ++i)
                a[i] /= n;
}
```

Additional optimization

Here are a list of other optimizations, which together can significantly speed up the above "improved" implementation:

- **Predposchitat reverse bits** for all numbers in a global table. It is especially easy when the size $n$is the same for all calls.
- This optimization becomes noticeable when a large number of calls $fft()$. However, its effect can be seen even in the three calls (three calls - the most common situation, ie when it is required once to multiply two polynomials).
- Refuse to use vector(**go to normal arrays**).
- The effect of this depends upon the particular compiler, but is typically present and accounts for approximately 10% -20%.
- Predposchitat **all power** number $wlen$. In fact, in this cycle algorithm repeatedly made through all the powers of the number $wlen$of 0up to $len/2-1$:
- ```
            for (int i=0; i<n; i+=len) {
                base w (1);
  ```
  ```

```
                for (int j=0; j<len/2; ++j) {
                    [...]
                    w *= wlen;
                }
            }
```

- Accordingly, before this cycle we can predposchitat from an array all the required extent, and thus get rid of unnecessary multiplications in a nested loop.
- Tentative acceleration - 5-10%.
- Get rid of **array accesses in the indices** , instead use pointers to the current array elements, promoting their right to one at each iteration.
- At first glance, optimizing compilers should be able to cope with this, but in practice it turns out that the replacement of references to arrays $a[i+j]$and $a[i+j+len/2]$pointers in the program accelerates common compilers. Profit is 5-10%.
- **Abandon the standard type of complex numbers** $\mathrm{complex}$, rewriting it for your own implementation.
- Again, this may seem surprising, but even in modern compilers benefit from such rewriting can be up to several tens of percent! This indirectly confirms a widespread assertion that compilers perform worse with sample data types, optimizing work with them much worse than non-formulaic types.
- Another useful optimization is the **cut-off length** : when the length of the working unit becomes small (say, 4), to calculate the DFT for it "manually". If the paint in these cases explicit formulas with a length equal to $4/2$, the values of sines, cosines take integer values, whereby it is possible to get a speed boost for a few tens of percent.

Let us here describe improved realization with (except the last two items which lead to overgrowth of code):

```
int rev[MAXN];
base wlen_pw[MAXN];

void fft (base a[], int n, bool invert) {
    for (int i=0; i<n; ++i)
        if (i < rev[i])
            swap (a[i], a[rev[i]]);

    for (int len=2; len<=n; len<<=1) {
        double ang = 2*PI/len * (invert?-1:+1);
        int len2 = len>>1;
```

```
            base wlen (cos(ang), sin(ang));
            wlen_pw[0] = base (1, 0);
            for (int i=1; i<len2; ++i)
                  wlen_pw[i] = wlen_pw[i-1] * wlen;

            for (int i=0; i<n; i+=len) {
                  base t,
                        *pu = a+i,
                        *pv = a+i+len2,
                        *pu_end = a+i+len2,
                        *pw = wlen_pw;
                  for (; pu!=pu_end; ++pu, ++pv, ++pw) {
                        t = *pv * *pw;
                        *pv = *pu - t;
                        *pu += t;
                  }
            }
      }

      if (invert)
            for (int i=0; i<n; ++i)
                  a[i] /= n;
}

void calc_rev (int n, int log_n) {
      for (int i=0; i<n; ++i) {
            rev[i] = 0;
            for (int j=0; j<log_n; ++j)
                  if (i & (1<<j))
                        rev[i] |= 1<<(log_n-1-j);
      }
}
```

On common compilers faster than the previous implementation of this "improved" version of 2-3.

Discrete Fourier transform in modular arithmetic

At the heart of the discrete Fourier transform are complex numbers, roots n-th degree of unity. Its effective computation used features such roots as the existence of $n$different roots, forming a group (ie, the degree of the same root - always another square, among them there is one element - the generator of the group, called a primitive root).

But the same is true of the roots of $n$the second degree of unity in modular arithmetic. I mean, not for any module $p$there $n$different roots of unity, but these modules do exist. Still important for us to find among them a primitive root, ie:

$$(w_n)^n = 1 \pmod p,$$

$$(w_n)^k \neq 1 \pmod p, \qquad 1 \leq k < n.$$

All other $n - 1$roots n-th degree of unity in modulus $p$can be obtained as the degree of primitive root w_n(as in the complex case).

For use in the fast Fourier transform algorithm we needed to root primivny existed for some n, a power of two, and all the lesser degrees. And if in the complex case, there was a primitive root for anyone n, in the case of modular arithmetic is generally not the case. However, note that if $n = 2^k$, that is kth power of two, then modulo $m = 2^{k-1}$ have:

$$(w_n^2)^m = (w_n)^n = 1 \pmod p,$$

$$(w_n^2)^k = w_n^{2k} \neq 1 \pmod p, \qquad 1 \leq k < m.$$

Thus, if w_n- a primitive root $n = 2^k$-th degree of unity, w_n^2- a primitive root 2^{k-1}-th degree of unity. Therefore, all powers of two smaller n, the primitive roots of the desired extent also exist and can be calculated as corresponding degree w_n.

The final touch - for inverse DFT we used instead of $w_n$the inverse element: w_n^{-1}. But modulo a prime $p$inverse is also always there.

Thus, all the required properties are observed in the case of modular arithmetic, provided that we have chosen some rather large unit $p$and found it a primitive root$n$-th degree of unity.

For example, you can take the following values: module $p = 7340033$, $w_{2^{20}} = 5$. If this module is not enough to find another pair, you can use the fact that for the modules of the form $c2^k + 1$(but still necessarily simple) there will always be a primitive cube root $2^k$of unity.

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

Here, the function $\mathrm{reverse}$is the inverse of the $n$element modulo mod(see inverse element in the mod). Constants mod, specify the module and a primitive root, and - inverse element modulo .$\mathrm{root}\ \mathrm{root\_pw}\mathrm{root\_1}\mathrm{root}\mathrm{mod}$

As practice shows, the implementation of integer DFT works even slower implementation of complex numbers (due to the sheer number of transactions taking absolute value), but it has advantages such as low memory usage and the lack of rounding errors.

Some applications

In addition to direct application for multiplying polynomials or long numbers, we describe here are some other applications of the discrete Fourier transform.

All possible sums

Problem: given two arrays $a[]$and $b[]$. You want to find all sorts of numbers of the form $a[i] + b[j]$, and for each of the number of prints the number of ways to get it.

For example, in $a = (1, 2, 3)$and $b = (2, 4)$obtain 3 can be obtained by method 1, 4 - one and 5 - 2, 6 - 1 7 - 1.

Construct the arrays $a$and $b$two polynomials $A$and B. As in the polynomial degrees will be performing numbers themselves, ie values $a[i]$($b[i]$), and as the coefficients when they - by how many times that found in the array a(b).

Then, multiplying these two polynomials for $O(n \log n)$, we obtain a polynomial C , where the number of degrees will be all kinds of species $a[i] + b[j]$, and their coefficients are just the required number of

All kinds of scalar products

Given two arrays $a[]$and $b[]$one length n. You want to display the values of each of the scalar product of the vector $a$for the next cyclic shift vector b.

Invert the array $a$and assign it to the end of the $n$zeros, and the array b- simply assign himself. Then multiply them as polynomials. Now consider the coefficients of the product $c[n \dots 2n - 1]$(as always, all the indexes in the 0-indexed). We have:

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

Since all the elements $a[i] = 0,\ i = n \dots 2n - 1$, we get:

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

It is easy to see in this sum, it is the scalar product $a$on $k - n - 1$the first cyclic shift. Thus, these coefficients (starting with $n - 1$the first and pumping $2n - 2$th) - is the answer to the problem.

The decision came with the asymptote $O(n \log n)$.

Two strips

Two strips are defined as two Boolean (ie numerical values 0 or 1) of the array $a[]$and $b[]$. Want to find all such positions in the first strip that if you apply, starting with this position, the second strip in any place will not work trueimmediately on both strips. This problem can be reformulated as follows: given a map strips as 0/1 - you can get up into the cell or not, and given some figure as a template (as an array, where 0 - no cells, 1 - yes), requires Find all positions in the strip which can be attached figure.

This problem is actually no different from the previous problem - the problem of the scalar product. Indeed, the dot product of two 0/1 arrays - the number of elements in which both units were. Our task is to find all cyclic shifts of the second strip so that there was not a single item, which would be in both strips were unity. Ie we have to find all cyclic shifts of the second array, in which the scalar product is zero.

Thus, this problem we decided for $O(n \log n)$.