added: 10 Jun 2008 19:04
Edited: 9 Nov 2012 12:38

# Fast Fourier transform of 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. It is clear that the multiplication of two long numbers can be reduced to a multiplication of polynomials, so the two long numbers can also multiply during $O(n \log n)$.

The invention Fast Fourier Transform attributed Cooley (Coolet) and Taki (Tukey) - 1965 Actually FFT repeatedly invented before, but its importance was not fully realized until the advent of modern computers. Some researchers credited with the discovery of the FFT Runge (Runge) and König (Konig) in 1924 Finally, the discovery of this method is attributed to more Gauss (Gauss) in 1805

**Contents** [hide]



## Discrete Fourier Transform (DFT)

Suppose there is a polynomial $n$of degree th:

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

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 by setting them equal to zero.

Of the theory of functions of a complex variable is known that the complex roots $n$of unity th exists exactly $n$. 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 of $n$th roots of unity) is such that all 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 of the vector of its coefficients $(a_0, a_1, \ldots, a_{n-1})$) are the values of the polynomial at the points $x = w_{n,k}$, ie, is a vector:

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

Defined similarly 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 the DFT coefficients of the polynomial to its values in the complex roots of $n$th roots of unity, the inverse DFT - on the contrary, from the values of the coefficients of the polynomial recovers.

## The use 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 polynomials.

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

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

But it does mean 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 but a 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 to calculate the DFT and inverse DFT of the time $O(n \log n)$, then the product of two polynomials (and, consequently, the two long numbers) we can find for the same asymptotic behavior.

It should be noted that, firstly, the result should be two polynomials of degree one (simply adding the coefficients of one of these zeros). Secondly, as a result of 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 coefficients equal to zero).

## Fast Fourier Transform

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

The basic idea is to divide the FFT coefficient vector into two vectors, the recursive computation of the DFT for them, and the union results in 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 $A_1$have 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 $\mathrm{DFT}(A)$, then we obtain 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 $\{y_k^1\}_{k=0}^{n/2-1} = \mathrm{DFT}(A_1)$. Let us find the expression for $\{y_k\}_{k=0}^{n-1} = \mathrm{DFT}(A)$.

Firstly, 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 to calculate the total vector $\{y_k\}$:

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

(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.

## Inverse FFT

So, let a vector $(y_0, y_1, \ldots, y_{n-1})$- the values of a polynomial $A$of degree $n$at points $x = w_n^k$. Need to recover the coefficients $(a_0, a_1, \ldots, a_{n-1})$of the polynomial. This well-known problem is called **interpolation** , for this task, there are some common algorithms for the solution, but in this case will be obtained by a very simple algorithm (a simple fact that it is virtually identical to the 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 $(a_0, a_1, \ldots, a_{n-1})$can be found by multiplying the vector $(y_0, y_1, \ldots, y_{n-1})$by the inverse matrix to the matrix, which stands on the left (which, incidentally, is called a Vandermonde matrix):

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

A direct check 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 observe that these two tasks 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 $w_n^k$should 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 it can also be performed during the time $O(n \log n)$.

## Implementation

Consider a simple recursive **implementation of the FFT** and IFFT, implement them in a single function, as the difference between direct and inverse FFT minimal. For storing complex numbers using the standard in 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 of $\mathrm{a}$the function is passed the input vector of coefficients in the same and it will contain the result. Argument $\mathrm{invert}$ shows direct or inverse DFT should be calculated. Inside the function first checks if the vector length $\mathrm{a}$is equal to one, there is nothing else to do - he is the answer. Otherwise, the vector $\mathrm{a}$is split into two vectors $\mathrm{a0}$and $\mathrm{a1}$for which recursively calculated DFT. Then we calculate the value of $w_n$, and the plant variable $w$containing the current degree $w_n$. Then calculated the elements of the result of the DFT on the above formulas.

If the flag is specified $\mathrm{invert} = \mathrm{true}$, then $w_n$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, the result just happens 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 feature works with polynomials with integer coefficients (although, of course, in theory there is nothing stopping her work with fractional coefficients). However, it appears the problem of a large error in the calculation of DFT: the error can be significant, so the rounding of the best 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 polynomials should be normalized, ie, perform all transfers 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 is never surpass the total length of the number, the size of the vector $\mathrm{res}$will 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 the even positions attributed to the same time to create a vector, and on the odd - to another. However, if we re-ordered elements in a certain way, the need for creating temporary vectors would then be eliminated (ie, all the calculations we could produce "in situ", 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 of positions which are equal to one - to the vector $a_1$. At 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 $a$according to the new indexes, we obtain the desired order (it is called a **bitwise inverse permutation** (bit-reversal permutation)).

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

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

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, in the bitwise inverse permutation, this corresponds to a separation of the vector into two halves: the first $n/2$element and the last $n/2$element. Then there is a recursive call on each half; let the resulting DFT of each of them was returned in place of the elements themselves (ie, the first and second halves of the vector $a$, respectively):

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

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

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

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:

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

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

le 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 hold for all other levels of recursion. Thus, **after applying the bitwise inverse permutation to calculate the DFT can be on the spot** , without any additional arrays.

But now you can **get rid of the recursion** explicitly. So, we applied bitwise inverse permutation elements. Now do all the work being done by the lower level of recursion, ie vector $a$divide into pairs of elements for each applicable transformation of butterflies, resulting in the vector $a$will be the results of the lower level of recursion. In the next step the vector divide $a$at quadruple elements applied to each butterfly transform, thus obtaining a DFT for every fours. 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 need to be exchanged (unless this condition, each couple will exchange twice, and in the end nothing will happen).

Then, the $\lg n - 1$algorithm steps, on $k$th 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 $\text{wlen}$. Cycle through $i$terated by block, and invested in it by the cycle $j$applies the transformation to all elements of the butterfly unit.

You can perform further **optimizations reverse bits** . In the previous implementation, we obviously took over all bits of the number, simultaneously building bitwise inverted number. However, reversing the bits can be performed in a different way.

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

We give a list of other optimizations, which together can significantly speed up the preceeding "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, the effect of it 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 $\mathrm{vector}$( **go to normal arrays** ).

  The effect of this depends upon the particular compiler, but typically it is present and approximately 10% -20%.

- Predposchitat **all power** numbers $wlen$. In fact, in this cycle of the algorithm repeatedly made the passage in all degrees of $wlen$on $0$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 some array all the required power, and thus get rid of unnecessary multiplications in the nested loop.

  Tentative acceleration - 5-10%.

- Get rid of **references to arrays in the indices** , instead use pointers to the current array elements, promoting their right to 1 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 to accelerate the program in popular compilers. Prize 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 a 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 you paint these cases in the form of explicit formulas for a length equal to $4/2$, the values of the sine-cosine take integer values, due to which you can get a speed boost for another few tens of percent.

Here we present the realization of the described improvements (except the last two items which lead to the proliferation of codes):

```
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, this implementation faster than the previous "improved" version of the 2-3.

## The discrete Fourier transform of the modular arithmetic

At the heart of the discrete Fourier transform are complex numbers, roots $n$th roots of unity. To effectively calculate it 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 $n$th roots of unity in modular arithmetic. Well, not for any module $p$there exists $n$a variety of roots of unity, but these modules do exist. Is still important for us to find among them a primitive root, ie .:

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

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

For use in the fast Fourier transform algorithm we needed to primivny root existed for some $n$, a power of two, as well as 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 $n = 2^k$, ie $k$Star power of two, the modulo $m = 2^{k-1}$have:

$$\begin{aligned}(w_n^2)^m = (w_n)^n &= 1 \pmod p,\\ (w_n^2)^k = w_n^{2k} &\neq 1 \pmod p, \qquad 1 \le k < m.\end{aligned}$$

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

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

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 to be a primitive root $n$th roots 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 is always 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 $n$an element modulo $\mathrm{mod}$(see. inverse element in the mod ). Constants $\mathrm{mod}$, determine the module and a primitive root, and - the inverse of an 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 huge number of operations modulo), but it has advantages such as low memory usage and the lack of rounding errors.

## Some applications

In addition to direct application to multiply 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 species $a[i] + b[j]$, 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 the number of 3 may be prepared 1 by the method, 4 - and one, 5 - 2, 6 - 1 7 - 1.

Construct from the arrays $a$and $b$two polynomials $A$and $B$. As the degree of the polynomial will be performing numbers themselves, ie values $a[i]$( $b[i]$), and as the coefficients of them - the number of times it occurs in the array by $a$( $b$).

Then, multiplying these two polynomials in $O(n \log n)$, we get a polynomial $C$, where the powers are all sorts 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 $b[]$the same length $n$. You want to display the values of each of the inner 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 indices 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 $k - n - 1$th cyclic shift. Thus, these coefficients (since $n - 1$th and pumping $2n - 2$of th) - is the answer to the problem.

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

## Two strips

Given two strips defined as two Boolean (ie numeric with 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 $\mathrm{true}$ right on both strips. This problem can be reformulated as follows: given a map of the strip, as 0/1 - you can get up into the cell or not, and has some figure as a template (in the form of an array, in which 0 - no cells, 1 - yes), requires find all the positions in the strip, which can be attached figure.

This problem is in fact no different from the previous problem - the problem of the scalar product. Indeed, the dot product of two arrays 0/1 - the number of elements in which both were unity. Our task is to find all cyclic shifts of the second strip so that there was not a single element, which would be in both strips were one. 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 $O(n \log n)$.

---

Непонятно, в чем вопрос. Асимптотика алгоритма - O(N log N) действий, а чтобы говорить о скрытой в асимптотике константе, надо договориться о том, что именно мы считаем за одно действие. Если вас интересует этот вопрос с практической стороны, то тут не может быть ничего лучше, кроме как запустить эту реализацию на конкретном компьютере и померять время её работы.

• Ответить • Поделиться ›

**Samojlov Valera** • год назад

Макс, "не что иное" o_O

• Ответить • Поделиться ›

**Дима Ярусевич** • год назад

Здравствуйте.

Я не совсем понимаю 1 момент, возможно, он очевиден, но тем не менее.
В этой строке res[i] = int (fa[i].real() + 0.5); почему именно real() берется, а не abs()(модуль числа) скажем.

Спасибо.

• Ответить • Поделиться ›

**anonymous** • 2 года назад

При замене complex<double> на собственную структурку комплексных чисел выигрыш составляет на самом деле не несколько десятков процентов, а примерно 250-300%. Думаю, можно поменять в статье complex<double> на рукописные комплексные числа. Также для любителей Java: не используйте собственный класс Complex, а передавайте в функцию два массива real[] и imag[]. Это увеличит производительность более чем в 10 раз. Есть и минус такого подхода: надо расписывать умножение комплексных чисел, но по сравнению с плюсом такого подхода он ничтожен.

• Ответить • Поделиться ›

**e_maxx** Модератор → anonymous • 2 года назад

Любители Java, наверное, и так знают, что обречены отказываться от любых структур-объектов во всех критичных местах :)

• Ответить • Поделиться ›

**Andrey Grigoriev** • 2 года назад

А ещё можно вообще выбросить реверс битов, если комбинировать БПФ с прореживанием по частоте и с прореживанием по времени...

• Ответить • Поделиться ›

**Andrey Grigoriev** • 2 года назад

Могу предложить ещё одну оптимизацию... Для действительных сигналов можно за один проход БПФ вычислить два Фурье-образа. HINT: Фурье-образ действительного сигнала -- чётная функция, образ мнимого сигнала -- нечётная. Таким образом, запихиваем коэффициенты первого числа в действительную часть, а второго -- в мнимую. Функцию, полученную в Фурье-области, разбиваем на чётную (образ 1-го числа) и нечётную (образ 2-го числа) составляющие.

При использовании Фурье в суррогатных полях и чисто бинарном представлении (как в 2^n-ичной системе) можно брать по модулю 2^k -- тогда можно избавиться от делений по модулю (оно будет проходить за счёт переполнения разрядов), но тогда 2^k различных корней из 1 уже не будет, и само быстрое Фурье будет не порядка степени двойки, и будет медленнее и гораздо муторнее в реализации...

• Ответить • Поделиться ›

**Enigma** • 2 года назад

Правильно ли я понимаю, что при реализации этого алгоритма в длинной арифметике придётся сначала посчитать число PI, с той же точностью каким-то другим алгоритмом, а потом просчитать заодно и синусы с косинусами?

• Ответить • Поделиться ›

**e_maxx** Модератор → Enigma • 2 года назад

Насколько я понимаю, да. Возможно, если ситуация позволяет, будет проще перейти к БПФ по простому модулю.

• Ответить • Поделиться ›

**Alexander Fedulin** • 2 года назад

В примере целочисленного ДПФ по модулю, примитивный корень должен быть 3.

• Ответить • Поделиться ›

**Alexander Fedulin** → Alexander Fedulin • 2 года назад

А, нет, понятия перепутал.

• Ответить • Поделиться ›

**Sairus** • 2 года назад

нашел ошибку в предпоследней версии программы в строчке
double ang = 2*PI/len * (invert?-1:+1);прямое преобразование - степень отрицательна, обратное преобразование - положительна, не так ли?

надо поменять -1 и 1

• Ответить • Поделиться ›

**e_maxx** Модератор ➚ Sairus • 2 года назад

Почему же, при прямом преобразовании степень положительна, при обратном - отрицательна. Да и эта строчка одинакова во всех приведенных реализациях.

• Ответить • Поделиться ›

Sairus ➚ e_maxx • 2 года назад

советую пересмотреть литературу, ну или хотя бы википедию: http://ru.wikipedia.org/wiki/Д...

• Ответить • Поделиться ›

**e_maxx** Модератор ➚ Sairus • 2 года назад

Я не улавливаю, почему на приведенной вами странице в Википедии (и в её английском варианте, с которого она, очевидно, списывалась) использовались отрицательные степени. На другой странице (http://ru.wikipedia.org/wiki/Б... ) степени уже используются положительные.

Замечу, что в книге Кормена (перевод, 2-е изд., стр. 935-938) таки используются положительные степени, и, очевидно, доверие к книге гораздо выше, чем к Википедии.

2 • Ответить • Поделиться ›

Sairus ➚ e_maxx • 2 года назад

по вашей ссылке положительная степень появляется только в основном алгоритме, где это не суть важно. при выводе преобразования из ДФП стоит уже отрицательная степень.
а вообще на википедии пишут еще:
"Разные источники могут давать определения, отличающиеся от приведённого выше выбором коэффициента перед интегралом, а также знака «−» в показателе экспоненты. "

Но в большинстве источников (в т.ч. в книгах Дженкинса и Ваттса "Спектральный анализ", Фрэнкса "Обработка сигналов", в справке MATLAB) дается определение со знаком "-" в экспоненте для прямого преобразования.

Мне кажется, что это зависит от области применения ПФ.
В физике (в частности, при обработке сигналов) везде используется формула с "-", так как с ней удобнее работать.Возможно, если преобразование используется только для упрощения математических вычислений, могут быть использованы другие формулы.

• Ответить • Поделиться ›

Andrei ➚ Sairus • 2 года назад

Прямое дискретное преобразование Фурье - степень положительная, а обратное - отрицательная, поскольку множитель W = exp(-i*2*pi/N). При этом раскрыв формулы Прямого и Обратного ДПФ