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

# Fast Fourier transform of 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 $n$ of time $O(n \log n)$, which is much better time $O(n^2)$, achieved a trivial multiplication algorithm. It is clear that the multiplication of two long numbers can be reduced to the multiplication of polynomials, so the two long numbers also can be multiplied over time $O(n \log n)$.

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

**Contents** [hide]


- Fast Fourier transform of O (N log N). Application to the multiplication of two polynomials or long numbers
  - The discrete Fourier transform (DFT)
  - The use of DFT for the rapid multiplication of polynomials
  - Fast Fourier transform
  - Inverse FFT
  - Implementation
  - An improved: computing "on the spot" with no additional memory
  - Additional optimization
  - Discrete Fourier transform in modular arithmetic
  - Some applications
    - All sorts of sums
    - All kinds of scalar products
    - Two strips


## The 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 by setting them equal to zero.

From the theory of functions of a complex variable is known that the complex roots of $n$the second 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 $n$-degree of unity) is such that all the other roots are its 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 a polynomial at points $x = w_{n,k}$, ie it is a vector:

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

Similarly defined and **the inverse discrete Fourier transform** (InverseDFT). The inverse DFT to the vector of values 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 DFT changes in the coefficients of the polynomial to its values in the complex roots of $n$the second degree of unity, the inverse DFT - on the contrary, the values of the coefficients of the polynomial recovers.

## The use of DFT for the ra

Given two polynomials $A$and $B$. Calculate alues of polynomials.

Now, what happens when the multiplicatio e.

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

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, we get nothing, as the DFT of the polynomial $A \times B$:

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

Finally, applying the inverse DFT, we get:

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

where, again, right under the product of two DFT mean pairwise products of the elements of the vectors. Such work is obviously required to calculate only $O(n)$operations. Thus, if we learn how to calculate the DFT and inverse DFT for 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 the first two polynomials to be lead to the same degree (just adding the coefficients of one of these zeros). Second, as a result of the product of two polynomials of degree $n$obtained by a polynomial of degree $2n - 1$, 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 of the time $O(n \log n)$. This method is based 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 combining the results into a single FFT.

So, let there be a polynomial of $A(x)$degree $n$, where $n$- the 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 check 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 computed $\mathrm{DFT}(A_0)$ and $\mathrm{DFT}(A_1)$calculated $\mathrm{DFT}(A)$, and we obtain the desired fast Fourier transform (since it is a standard chart of "divide and conquer", and is known for its 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 $\{y_k^1\}_{k=0}^{n/2-1} = \mathrm{DFT}(A_1)$. We 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 the calculation of the whole 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 $a - bc$, sometimes referred to as "the transformation of the butterfly" ("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 $x = w_n^k$. You need to recover the coefficients of $(a_0, a_1, \ldots, a_{n-1})$the polynomial. This well-known problem is called **interpolation** , for this problem, there are common solution algorithms, but in this case it will be received very simple algorithm (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 $(a_0, a_1, \ldots, a_{n-1})$ can be found by multiplying the vector $(y_0, y_1, \ldots, y_{n-1})$ by the inverse matrix to the matrix on the left (which, by the way, is called the 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 notice that these two problems are not real, so the coefficients $a_k$ can be found in the same algorithm of "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 hardly differs from direct DFT calculations, and it can also perform a time $O(n \log n)$.

## Implementation

Consider the following simple recursive **implementation of the FFT** and IFFT sell them as one function, because the differences between the direct and inverse FFT low. For storage of complex numbers we use 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];
```

```
                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 $\mathrm{a}$passed to the function input vector of coefficients, and it will contain the same result. The argument $\mathrm{invert}$shows direct or inverse DFT to be calculated. Inside the function first checks if the length of the vector $\mathrm{a}$is unity, 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 the DFT is calculated recursively. It then calculates the value $w_n$and the plant variable $w$containing the current level $w_n$. Then calculated the resulting elements DFT on the above formulas.

If the flag is specified $\mathrm{invert} = \mathrm{true}$, it $w_n$is replaced by $w_n^{-1}$, and each element of the result is divided by 2 (noting that the division by 2 will occur at each level of recursion, then eventually just turns out that all the items on the share $n$).

Then the function for **multiplying two polynomials** will be 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, the problem here is shown a large error in the DFT: error can be significant, so it is better to round the number of the 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 virtually no different from the function to multiply polynomials. The only feature - that after the multiplication of numbers as polynomials should be normalized, 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;
        }
```

(Because the length of the product of two numbers will never exceed the total length of numbers, then the size of the vector $\mathrm{res}$will be enough to fulfill all the carries.)

## An improved: computing "on the spot" with no additional memory

To increase the efficiency abandon recursion explicitly. In the above recursive implementation, we obviously shared the vector $\mathrm{a}$into two vectors - the elements in the even positions attributed to the same time creates a vector, and on the odd - to another. However, if we re-ordered items in a certain way, the need for a temporary vectors would then be eliminated (ie, all of 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 of the positions which are equal to one - to the vector $a_1$. The second level of recursion is performed the same but for the second bits

the positions which are equal to one - to the vector $a[1]$. This second level of recursion is performed this same but for the second bits, etc. So if we are in the position $i$ of each element $a[i]$ invert the order of the bits, and reorder elements in an array $a$ according to the new index, and then we get the desired order (it is called a **bit-reverse permutation** (bit-reversal permutation)).

For example, $n = 8$ this procedure 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, at the first level of recursion (surrounded by braces) recursive algorithm conventional separation occurs 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 bit-reverse 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 of each half, and let the resulting DFT of each of them was returned in place of the elements themselves (ie, in 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 need to merge the 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 thus that it should have 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 get:

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

That is, We got exactly the desired DFT of the vector $a$.

We have described the process of computing the DFT at the first level of recursion, but it is clear that the same arguments hold for all other levels of recursion. Thus, **after applying the bit-reverse 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 have applied the bit-reverse permutation of the elements. Now do all the work being done by the lower level of recursion, ie vector $a$ divide into pairs of elements for each conversion applicable butterfly, resulting in the vector $a$ will be the results of a lower level of recursion. In the next step the vector divide $a$ by four elements, the transformation applied to each butterfly obtain a result for each DFT 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 get 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, the vector $a$is applied bit-reverse permutation, which count the number of significant bits ( $\lg n$), among $n$, and for each position $i$is the corresponding position, the bit record which has the bit representation of the number $i$recorded in the reverse order. If, the resulting position was more $i$, the elements in these two positions should be exchanged (unless it is a condition that each pair will exchange twice, and in the end nothing will happen).

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

You can further **optimize the reverse bits** . In the previous implementation, we obviously took over all bits of the number, at the same bit-inverted order number. However, the reverse bits can be done in a different way.

For example, suppose $j$- is counted by a number equal to the inverse number of 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" notation. In a conventional binary add one - then remove all of the 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 the bit 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 get 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 is a list of other optimizations, which together can significantly speed up the above "enhanced" implementation:

- **Predposchitat reverse bits** for all numbers in a global table. This is especially easy when the size $n$is the same for all calls.

  This optimization becomes significant 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, that is when you need a time to multiply two polynomials).

- Refuse to use $\mathrm{vector}$( **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 degrees of** numbers $wlen$. In fact, this algorithm loop repeatedly performed by extending all the powers of the number $wlen$of $0$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 from an array all the required extent, and thus get rid of unnecessary multiplications in a nested loop.

  Approximate acceleration - 5-10%.

- Get rid of the **array accesses in the indices** , instead use pointers in the array, moving them to the right 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 to speed up the program to the common compilers. The payoff is 5-10%.

- **Abandon the standard type of complex numbers** $\mathrm{complex}$ by rewriting it on its 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 the assertion of conventional wisdom that compilers perform worse with sample data types, optimizing work with them much worse than non-formulaic types.

- Another useful optimization is to **cut off the length** , when the length of the working unit is small (say, 4), to calculate the DFT for it "by hand." 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, whereby it is possible to get speed boost for a few tens of percent.

We present herein described improvements to implementation (except for the last two points, which lead to overgrowth 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 the sales of common compilers faster than the previous "improved" version of 2-3.

## Discrete Fourier transform in modular arithmetic

At the heart of the discrete Fourier transform are complex numbers, the roots of $n$the second degree of unity. To effectively used in its calculation of roots of such features as the existence $n$of different roots, forming a group (ie, the degree of the same root - always another root, and among them there is one element - the generator of the group, called the primitive root).

But the same is true of the roots of $n$the second degree of unity in modular arithmetic. More precisely, not for any module $p$can find $n$a variety of roots of unity, however, these modules do exist. Still it is important to find a primitive root of them, i.e.:

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

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

For use in the fast Fourier transform algorithm we needed to get to the root of primivny 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, it is generally not the case. However, note that if $n = 2^k$, that is $k$th power of two, then modulo $m = 2^{k-1}$have:

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

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

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

Thus, all the necessary properties observed in the case of modular arithmetic, provided that we have chosen some fairly large unit $p$ , and found in it a primitive root of $n$the second degree of unity.

For example, you can take the following values: a 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) is always a primitive 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 * 111 * wlen % mod);
                        }
                }
        }
        if (invert) {
                int nrev = reverse (n, mod);
                for (int i=0; i<n; ++i)
                        a[i] = int (a[i] * 111 * nrev % mod);
        }
}
```

This function $\mathrm{reverse}$is the inverse of $n$an element modulo $\mathrm{mod}$(see Return the item in the mod ). The constants $\mathrm{mod}$, determine the module and a primitive root, and - the inverse of the element in absolute value .$\mathrm{root}\ \mathrm{root\_pw}\mathrm{root\_1}\mathrm{root}\mathrm{mod}$

As practice shows, the implementation of integer DFT works even slower implementation of complex numbers (due to the huge number of transactions taking the absolute value), but it has advantages such as less use of memory and no rounding errors.

## Some applications

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

### All sorts of sums

The problem: given two arrays $a[]$and $b[]$. Required 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, $a = (1, 2, 3)$and $b = (2, 4)$obtain 3 can be obtained by method 1, 4 - and one, 5 - 2, 6 - 1 7 - 1.

We construct the array $a$and $b$two polynomials $A$and $B$. As the degrees of the polynomial will be performing numbers themselves, ie value $a[i]$( $b[i]$), and as the coefficients of them - it is many times encountered in the array by $a$( $b$).

Then, multiplying these two polynomials over $O(n \log n)$, we get a polynomial $C$, where the number of degrees will be 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 $b[]$one length $n$. You want to display the value of each 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 $c[n \ldots 2n - 1]$(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 $a[i] = 0,\ i = n \ldots 2n - 1$, then we obtain

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

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

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

### Two strips

Given two strips defined as two Boolean (ie, numerical values of 0 or 1) of the array $a[]$and $b[]$. Required to find all such positions in the first strip that if you make from this position, the second strip, in any place will not work $\mathrm{true}$right away on both strips. This problem can be reformulated as follows: given a map of the strip, in the form of a 0/1 - you can get up in the cage or not, and given some figure as a template (in the form of an array in which 0 - no cells, 1 - yes), requires find the position 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 scalar product of two 0/1 arrays - 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 item, which would be in both strips were one. That is, we have to find all cyclic shifts of the second array, in which the dot product is zero.

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

---

27 комментариев ★ 9

Оставить сообщение...

Лучшие Сообщество Поделиться

witua • 6 месяцев назад

При умножения, например, многочлена (10^4, 10^4, ..., 10^4) на (10^4, 10^4, ..., 10^4), степени которых также 10^4, в любой из заданных реализаций получается серьезная погрешность. Это происходит по причине постоянного умножения w *= wlen и накопления погрешности, стоит, я считаю, каждый раз пересчитывать угол ang_cur = (2*PI*i/n) * (invert ? -1 : 1) и делать w = base(cos(cur_ang), sin(cur_ang)).

6 Ответить Поделиться ›

Andrey Petrov • 9 месяцев назад

Удивительное рядом: int(d + 0.5) округляет до ближайшего целого только положительные числа, при d = -1.0 ответом будет 0.

1 Ответить Поделиться ›

e_maxx Модератор → Andrey Petrov • 9 месяцев назад

Верно, и на это уже натыкались "невнимательные читатели" :)
Добавил фразу про то, что это может работать неправильно; просто в рамках статьи рассматривались только многочлены с неотрицательными степенями.

1 Ответить Поделиться ›

Andrey Petrov → e_maxx • 9 месяцев назад

Неотрицательными коэффициентами?
Я не нашёл упоминания о том, что рассматриваются только они, поэтому, кажется, даже внимательные могли ошибиться.

1 Ответить Поделиться ›

e_maxx Модератор → Andrey Petrov • 9 месяцев назад

UPD. Действительно, был не прав :) Про неотрицательность коэффициентов ничего не утверждалось.

Ответить Поделиться ›

Victor N Yegorov • год назад

а не могли бы вы выделять то, что вы обновляете в статье и возможность отключать это при необходимости.

1 Ответить Поделиться ›

e_maxx Модератор → Victor N Yegorov • год назад

К сожалению, пока история изменений статей никак не хранится.

Ответить Поделиться ›

Oleg Kovalov • год назад

А какова константа сложности в первой версии программы?

1 Ответить Поделиться ›

e_maxx Модератор → Oleg Kovalov • год назад

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 • 4 месяца назад

Макс, "не что иное" o_O

Ответить Поделиться ›

Дима Ярусевич • 5 месяцев 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 • 8 месяцев 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.</double></double>

Ответить Поделиться ›

e_maxx Модератор ➔ anonymous • 8 месяцев назад

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

Ответить Поделиться ›

Andrey Grigoriev • 9 месяцев назад

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

Ответить Поделиться ›

Andrey Grigoriev • 9 месяцев 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 • 11 месяцев 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 • 11 месяцев назад

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

Ответить Поделиться ›

Alexander Fedulin • год назад

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

Ответить Поделиться ›

Alexander Fedulin ➔ Alexander Fedulin • год назад

А, нет, понятия перепутал.

Ответить Поделиться ›

Sairus • год 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 • год 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 • год назад

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

Ответить Поделиться ›

e_maxx Модератор ➚ Sairus • год 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 • год 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 • год назад

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