Fast Fourier Transform

Andrew Stankevich

Barcelona, October 1, 2018

1 Introduction

1.1 Complex Numbers and Roots of Unity

The basic knowledge of complex numbers is required. If you are not familiar with complex numbers, please read about them before the lecture. The short summary follows.

Let $i = \sqrt{-1}$. The set $\mathbb{C}$ of complex numbers consists of all numbers of the form $a + bi$. Complex numbers form a field: they can be added, subtracted, multiplied and divided. For $x = a + bi$ its conjugate is $\overline{x} = \overline{a + bi} = a - bi$.

C++ standard header `<complex>` has `complex<T>` type which is usually used as `complex<double>`.

The value $|a + bi| = \sqrt{a^2 + b^2}$ is called the absolute value of the complex number. If we divide the complex number $a + bi$ by its absolute value r we get $a' + b'i$ where $a' = a/r$, $b' = b/r$. Now $a'^2 + b'^2 = 1$, so $a' = \cos\varphi$ and $b' = \sin\varphi$ for some φ. This value is called the phase of the complex number. If $a + bi = r(\cos\varphi + i\sin\varphi)$ we write $a + bi = re^{i\varphi}$. If $a + bi = re^{i\varphi}$ and $c + di = se^{i\psi}$, then $(a + bi)(c + di) = (ac - bd) + (ad + bc)i = rse^{i(\varphi+\psi)}$. If $x = re^{i\varphi}$, then $x^n = r^n e^{in\varphi}$.

Since $e^{2\pi i} = \cos 2\pi + i\sin 2\pi = 1$, it is $e^{i(\varphi+2\pi)} = e^{i\varphi}$.

The n-th primitive root of unity ω_n is the value such that $\omega_n^n = 1$, and $\omega_n^k \neq 1$ for any $1 \leq k < n$. For real numbers there are no primitive roots of unity except for $n = 1$ and $n = 2$, but for complex numbers there are primitive roots of unity for any n. The value $\omega_n = e^{2\pi i/n}$ is the n-th primitive root of unity.

If $n = km$ and ω_n is the n-th primitive root of unity, ω_n^k is the m-th primitive root of unity.

1.2 Divide and Conquer Algorithms

The basic idea of divide and conquer algorithms is to divide their input to several parts, solve the problem for each part, and combine the result to the answer for the whole problem. A good example of a divide and conquer algorithm is merge sort algorithm.

The array $[a_0, a_1, \ldots, a_{n-1}]$ must be sorted. Let $m = \lfloor n/2 \rfloor$, divide the array into two parts: $[a_0, a_1, \ldots, a_{m-1}]$ and $[a_m, a_{m+1}, \ldots, a_{n-1}]$. Sort each of them recursively. Now walk the two sorted arrays with two pointers and merge them into one sorted array. Initially point to the first elements of the two arrays. Each step put the smaller value to the resulting array, and move the corresponding pointer to the next element.

1.3 Generating Functions

Generating function of a sequence $[a_0, a_1, \ldots, a_n, \ldots]$ is a formal power series $A(t) = a_0 + a_1 t + a_2 t^2 + \ldots + a_n t^n + \ldots$.

For a finite array $[a_0, a_1, \ldots, a_{n-1}]$ its generating function is a polynomial $A(t) = a_0 + a_1 t + a_2 t^2 + \ldots + a_{n-1} t^{n-1}$.

2 Fast Fourier Transform

2.1 Discrete Fourier Transform

Discrete Fourier Transform (DFT) of an array $\mathbf{a} = [a_0, a_1, \ldots, a_{n-1}]$ is defined as follows. Let ω_n be the n-th primitive root of unity. Evaluate $A(t)$ for $t = \omega_n^0$, $t = \omega_n^1$, etc. Create an array $\mathbf{b} = [b_0, b_1, \ldots, b_{n-1}]$ where

$$b_j = A(\omega_n^j) = a_0 + a_1 \omega_n^j + a_2 \omega_n^{2j} + \ldots + a_{n-1} \omega_n^{(n-1)j} = \sum_{i=0}^{n-1} a_i \omega_n^{ij}.$$

We say that $\mathbf{b} = DFT(\mathbf{a})$.

Examples.

- $n = 1$, $\omega_1 = 1$, $DFT([a_0]) = [a_0]$.
- $n = 2$, $\omega_2 = -1$, $DFT([a_0, a_1]) = [a_0 + a_1, a_0 - a_1])$.
- $n = 4$, $\omega_4 = i$, $DFT([a_0, a_1, a_2, a_3]) = [a_0 + a_1 + a_2 + a_3, a_0 + a_1 i - a_2 - a_3 i, a_0 - a_1 + a_2 - a_3, a_0 - a_1 i - a_2 + a_3 i]$.

DFT is widely used in signal processing, but we will consider its usages in competitive programming.

There are two main points of view to DFT.

One way is to see DFT as representing a polynomial of the n-th degree as its values for n possible arguments. This representation allows to make arithmetic operations with polynomials easily: addition, subtraction correspond to element-wise addition, subtraction, correspondingly. So $DFT(\mathbf{a} \pm \mathbf{b}) = DFT(\mathbf{a}) \pm DFT(\mathbf{b})$.

If we consider two polynomials $A(t)$ and $B(t)$ and multiply them to get $C(t) = A(t)B(t)$, we see that $DFT(\mathbf{c})$ can be obtained by element-wise

multiplication of $DFT(\mathbf{a})$ and $DFT(\mathbf{b})$. A note: $\mathbf{c}$ contains the the number of elements equal to the number of elements in $\mathbf{a}$ and $\mathbf{b}$ together, so we must append the corresponding number of 0-s to the end of $\mathbf{a}$ and $\mathbf{b}$ if we are planning to multiply their generating polynomials.

The possibility to multiply polynomials represented as their DFT in $O(n)$ is one of the most important features of DFT used in competitive programming.

Another way to describe DFT is to consider $\mathbf{a}$ as an n-dimensional vector, and DFT as a linear transform. Thus DFT is specified by an $n \times n$ matrix F with $f_{ij} = \omega_n^{ij}$:

$$F = \begin{pmatrix} \omega_n^0 & \omega_n^0 & \dots & \omega_n^0 \\ \omega_n^0 & \omega_n^1 & \dots & \omega_n^{n-1} \\ \omega_n^0 & \omega_n^2 & \dots & \omega_n^{2(n-1)} \\ \vdots & \vdots & \ddots & \vdots \\ \omega_n^0 & \omega_n^{n-1} & \dots & \omega_n^{(n-1)^2} \end{pmatrix}.$$

This way $DFT(\mathbf{a}) = F\mathbf{a}$.

$$DFT(\mathbf{a}) = F\mathbf{a} = \begin{pmatrix} \omega_n^0 & \omega_n^0 & \dots & \omega_n^0 \\ \omega_n^0 & \omega_n^1 & \dots & \omega_n^{n-1} \\ \omega_n^0 & \omega_n^2 & \dots & \omega_n^{2(n-1)} \\ \vdots & \vdots & \ddots & \vdots \\ \omega_n^0 & \omega_n^{n-1} & \dots & \omega_n^{(n-1)^2} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_{n-1} \end{pmatrix}.$$

Note that F is a Vandermonde matrix, so it has some nice properties.

2.2 Fast Fourier Transform, Cooley-Tuckey Algorithm

To compute DFT efficiently we use divide and conquer algorithm very similar to merge sort. Let $n = 2^k$, if it is not, append the required number of zeroes to $\mathbf{a}$.

Consider $A(t) = a_0 + a_1 t + a_2 t^2 + \dots$. Rearrange its coefficients, first consider the even positions and then the positions: $A(t) = a_0 + a_2 t^2 + a_4 t^4 + \dots + a_1 t + a_3 t^3 + a_5 t^5 + \dots$. Now let us introduce two new polynomials, A_0 for even positions and A_1 for odd positions: $A_0(z) = a_0 + a_2 z + a_4 z^2 + \dots$ and $A_1(z) = a_1 + a_3 z + a_5 z^2 + \dots$, so

$$A(t) = A_0(t^2) + tA_1(t^2).$$

Since $\omega_n^2 = \omega_{n/2}$, we see that to get DFT for $A(t)$ we could take DFT for $A_0(t)$ and $A_1(t)$ and combine them in linear time like this:

```
def fft(a, n):
    if n == 1:
        return [a[0]]

    # split a to a_odd and a_even
    a_even = [a[0], a[2], ...]
    a_odd = [a[1], a[3], ...]

    # run fft recursively
    f_even = fft(a_even, n/2)
    f_odd = fft(a_odd, n/2)

    # reconstruct f values
    for i in 0 .. n/2-1:
        f[i] =     f_even[i] + ω_n^i * f_odd[i]
        f[i+n/2] = f_even[i] + ω_n^(i+n/2) * f_odd[i]

    return f
```

Each of A_0 and A_1 has the degree equal to $n/2$, so the time complexity is $T(n) = 2T(n/2) + O(n)$ and therefore $T(n) = O(n \log n)$.

The algorithm above was first described by Cooley and Tuckey, so it is sometimes called Cooley-Tuckey algorithm. The more common name is Fast Fourier Transform (FFT).

2.3 Implementation Tricks

The pythonish pseudocode above runs extremely slow, even if written in C++. There are two major issues and a couple of minor.

The first major issue is calculating ω_n^i. If we use `{cos(2 * pi * i / n), sin(2 * pi * i / n)}` we get a very slow code because cos and sin calculations are very expensive. The better way is to precalucalate them and store as `omega[i]`. If we need ω_m^i where $n = mk$ we use `omega[i * k]`.

```
vector<complex<double>> omega(n);
for (int i = 0; i < n; i++) {
    omega[i] = {cos(2*i*PI/n), sin(2*i*PI/n)};
}
```

This code can be optimized even more, by not making n cos and sin calls.

The second major issue are the recusive calls with array allocations. Let us consider a non-recursive implementation.

Let us rearrange the elements of the argument array to FFT, so that the first $n/2$ elements were the elements from even positions, and the last $n/2$

elements were the elements from odd positions. Now for each half (that we would recursively process) do the same, etc.

So the elements $[a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7]$ would be rearranged as $[a_0, a_2, a_4, a_6, a_1, a_3, a_5, a_7]$, then as $[a_0, a_4, a_2, a_6, a_1, a_5, a_3, a_7]$.

We see that the position $r(i)$ that the element from the position i ends at has the binary notation equal to the reverse of binary notation of i, prepended with zeroes to length $\log_2 k$. For example element a_4 from the position $4 = 100_2$ goes to the position $1 = 001_2$. We can calculate the array r using the following pseudocode:

```
r[0] = 0
for i in 1 .. n-1:
    r[i] = r[i / 2] / 2 + ((i & 1) << (logN - 1))
```

Now consider two recursive calls `fft` makes. The are both actually identical. So we can remove recursion and convert it to two (actually three, because the conquer step is still there) `for` cycles.

```
def fft(a, f):
  # reversed order
  for i in 0 .. n-1:
    f[i] = a[r[i]]

  for (k = 1; k < n; k = k * 2):
    for (i = 0; i < n; i = i + 2 * k):
      for j in 0 .. k-1:
        z = omega[j*n/(2*k)] * f[i + j + k]
        f[i + j + k] = f[i + j] - z
        f[i + j] = f[i + j] + z
```

This implementation allows some further optimizations, but they are not essential.

2.4 Inverse DFT

There are two approaches to inverse DFT.

The first one is to use the same procedure, but use $\omega_n^{-1} = \overline{\omega_n}$ instead of ω_n. It can be proved that $DFT_{\omega_n}(DFT_{\omega_n^{-1}}(\mathbf{a})) = n\mathbf{a}$, so after running inverse DFT you must divide the resulting array by n.

The second one uses the fact that $\omega_n^{n-i} = (\omega_n^{-1})^i$.

For an array $\mathbf{a} = [a_0, a_1, a_2, \ldots, a_{n-1}]$ denote as $\mathbf{a}^*$ the array $\mathbf{a} = [a_0, a_{n-1}, a_{n-2}, \ldots, a_1]$. So $DFT(DFT(\mathbf{a})^*) = n\mathbf{a}$.

To prove both formulas just substitute the definition of DFT and make some simplifications. Another way to prove it is to use properties of Vandermonde matrix.

3 FFT Usages

3.1 Polynomial Multiplication

Multiply polynomials $A(t)$ and $B(t)$. Append their coefficients by zeroes so that their length is power of 2 and is no smaller than the length of their product — 1 + sum of their degrees.

Run FFT for both polynomials, multiply resulting arrays element-wise, run inverse FFT and divide the result by n.

3.2 Big Integer Multiplication

Use digits of big integers as coefficients of the polygon (you can use bigger base than 10, for example 10^4). Multiply polynomials and normalize the result by simulating carry of the extra values to greater digits.

3.3 Usage in Dynamic Programming

Moving from one level of DP to another can sometimes be represented as polynomial multiplication (especially in combinatorics, where it often corresponds to generating functions multiplication).

Example: find the number of ways to represent n as the sum of m primes, order is important. Let $dp[i][j]$ be the number of ways to represent j as the sum of i primes. We have $dp[1][j] = 1$ iff j is prime, and $dp[i][j] = \sum_k dp[i-1][k] \cdot dp[1][i-k]$. We see that the formula for $dp[i][j]$ is exactly polynomial multiplication, if we make a polynomial $D_i(t)$ for $\mathbf{dp}[i]$, $D_i(t) = D_{i-1}(t) \cdot D_1(t)$.

An important improvement can be made here, if we don't need intermediate results, we can keep $D_i(t)$ represented as its DFT and get back to explicit polynomials only for $i = m$.

3.4 Usage in Pattern Matching

Consider two patterns of 0-s and 1-s s and t. We can append any number of spaces to the beginning of any of those. We want to align them in such way that the total number of equal values at the same position is maximized.

Reverse t and multiply them as polynomials. Now the k-th coefficient is equal to the number of i such that i-th element of s and $k-i$-th element of t reversed are both 1-s. Similarly find the corresponding number of matching 0-s. Now we can find the optimal alignment by finding the maximum.

4 Further Improvements

4.1 Modulo Implementation for Good Modules

Notice that we used FFT for complex numbers, not reals is that reals have no primitive roots of unity. What about modular arithmetics?

It turns out that sometimes there actually are primitive roots of unity for some modules. Namely there are the n-th primitive roots of unity if the module p is prime and n divides $p-1$. So if $p = t2^k + 1$, there are roots of unity for $n = 1, 2, 4, 8, \ldots, 2^k$. One notable such modulo often used in programming contests is $p = 998\,244\,353 = 7 \cdot 17 \cdot 2^{23} + 1$. It has primitive roots of unity for n being power of two up to $2^{23} = 8\,388\,608$.

To find the primitive roots of unity we will use g — the generator of the multiplicative group modulo p; g is the generator if $g^{p-1} = 1$, and for all q — prime divisors of $p-1$ it is $g^{(p-1)/q} \neq 1$. There are very many generators, so trying values in increasing order, or at random, will soon find one. After we find generator, $\omega_n = g^{(p-1)/n}$.

Now we can use modulo arithmetics instead of complex numbers arithmetics.

4.2 Using Imaginary Part

We usually have real numbers as input. We can use imaginary part of the complex number in the make two FFTs in one.

As before, for an array $\mathbf{a} = [a_0, a_1, a_2, \ldots, a_{n-1}]$ denote as $\mathbf{a}^*$ the array $\mathbf{a} = [a_0, a_{n-1}, a_{n-2}, \ldots, a_1]$, and denote as $\overline{\mathbf{a}}$ elementwise conjugate operation for $\mathbf{a}$.

Let $\mathbf{a} = \mathbf{b} + i\mathbf{c}$. Let $\mathbf{f}$, $\mathbf{g}$, and $\mathbf{h}$ the their DFT, respectively.

It is $f_i = g_i + ih_i$, $\overline{g_i} = g_{n-i}$, $\overline{h_i} = h_{n-i}$. So in order to extract $\mathbf{g}$ and $\mathbf{h}$ from $\mathbf{f}$, we can use the following formulas:

$$\mathbf{g} = (\mathbf{f} + \overline{\mathbf{f}^*})/2,$$

$$\mathbf{h} = (\mathbf{f} - \overline{\mathbf{f}^*})/(2i).$$

4.3 Modulo Implementation for Bad Modules

For bad modules p (not in the form of $x \cdot 2^k + 1$) we can make plain polynomial multiplication, and take the result modulo p after multiplication. However, there is a problem, the result of polynomial multiplication with coefficients of up to t can be up to $t^2 n$. Precision of `double` only allows to store integers approximitely up to 10^{15}, and rounding errors can also add up.

Solution: sqrt-decomposition, represent integers as pairs (x_{large}, x_{small}). Now the algorithm proceeds as follows:

```
- run ‘fft‘ on pairs ‘(a_small[i], a_large[i])‘
- run ‘fft‘ on pairs ‘(b_small[i], b_large[i])‘
- reconstruct ‘(f_small[i], f_large[i])‘ and ‘(g_small[i], g_large[i])‘
- let ‘h0 = f_small * g_small‘
- let ‘h1 = f_small * g_large + f_large * g_small‘
- let ‘h2 = f_large * g_large‘
- run inverse ‘fft‘ on ‘h0 + i * h1‘
- run inverse ‘fft‘ on ‘h2‘
```

Remarks

Special thanks to Vladimir Smykalov for his FFT presentation that was used when preparing these notes, Borys Minaiev, Ilya Zban, Vitaly Aksenov whose problems were used in the contest.