고급 3회차
Fast Fourier Transform in Problem Solving

서강대학교 전해성(seastar105)

What is Fast Fourier Transform?

FFT는 DFT를 빠르게 하는 알고리즘을 말합니다.

그럼 DFT는 뭐죠?

$$
\begin{aligned}
X_k &= \sum_{n=0}^{N-1} x_n \cdot e^{-\frac{i2\pi}{N}kn} \\
&= \sum_{n=0}^{N-1} x_n \cdot \left[\cos\left(\frac{2\pi}{N}kn\right) - i \cdot \sin\left(\frac{2\pi}{N}kn\right)\right],
\end{aligned}
\tag{Eq.1}
$$

Joseph Fourier

Convolution

DFT는 합성곱(Convoluton) 연산을 하는 데 쓰입니다.
그럼 또 합성곱 연산은 뭘까요?

$$a = [a_0, a_1, \ldots, a_{n-1}]$$
$$b = [b_0, b_1, \ldots, b_{n-1}]$$
$$c = a * b$$
$$c_k = \sum_{i+j=k} a_i b_j \ (0 \le i, j < n)$$

합성곱 연산의 기호는 *입니다.

길이가 n인 수열 a, b의 합성곱 c

c = a*b

Polynomial Multiplication

합성곱은 다항식의 곱과 형태가 같다!

$$
\begin{aligned}
a &= [a_0, a_1, \dots, a_{n-1}] \\
b &= [b_0, b_1, \dots, b_{n-1}] \\
c &= a * b \\
c_k &= \sum_{i+j=k} a_i b_j \ (0 \le i, j < n)
\end{aligned}
$$

$$
\begin{aligned}
f(x) &= \sum_{i=0}^{n-1} a_i x^i \\
g(x) &= \sum_{i=0}^{n-1} b_i x^i \\
h(x) &= f(x)g(x) \\
&= \sum_{k=0}^{2n-2} \Big(\sum_{i+j=k} a_i b_j \Big) x^k \\
&= \sum_{k=0}^{2n-2} c_k x^k
\end{aligned}
$$

합성곱의 시간복잡도는 $O(n^2)$입니다. FFT는 이를 O(n logn)으로 줄여줍니다.
일단, FFT라는 말은 잊어버리고 **n-1차 다항식 f(x), g(x)의 곱을 빠르게 합시다.**

Lagrange's Interpolation

Key Idea : **Lagrange's Interpolation**

n-1차 다항식은 n개의 서로 다른 점에서의 함숫값이 결정되면 유일하게 결정된다.

$$f(x) = a_0 + a_1x^1 + a_2x^2 + \ldots + a_{n-1}x^{n-1}$$

$$\updownarrow$$

$$(x_0, f(x_0)), (x_1, f(x_1)), (x_2, f(x_2)), \ldots, (x_{n-1}, f(x_{n-1}))$$

위 방식을 coefficient representation, 아래 방식을 point representation이라 합시다.
f(x)의 함숫값을 2n-2개, g(x)의 함숫값을 2n-2개 알고 있다고 하자.
그러면, h(x)=f(x)g(x)의 함숫값 2n-2개를 O(n)에 알 수 있습니다.

Still O(n^2)

Key Idea : **Lagrange's Interpolation**

함숫값 하나를 구하는 데에는 O(n)만큼 걸립니다.

그러면 함숫값 2n-2개 구하는 데에 O(n^2)이 걸립니다. ,따라서 coef -> point로 변환하는데 O(n^2)이 걸립니다. 그리고 point -> coef도 O(n^2)이 걸립니다.

$$f(x) = a_0 + a_1x^1 + a_2x^2 + \ldots + a_{n-1}x^{n-1}$$

$$\updownarrow$$

$$(x_0, f(x_0)), (x_1, f(x_1)), (x_2, f(x_2)), \ldots, (x_{n-1}, f(x_{n-1}))$$

nth Root of Unity

Another Key Idea : **nth Root of Unity**

$$z^n = 1 \;\; and \;\; z^m \neq 1 \; for \; m = 1, 2, 3, \ldots, n-1$$

위 조건을 만족하는 수를 nth root of unity라고 부르며 모든 자연수 n에 대해서 nth root of unity를 찾을 수 있습니다.

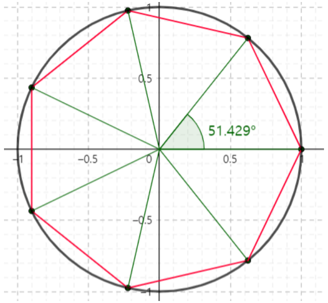
$$e^{\theta i} = \cos\theta + i\sin\theta$$

$$e^{2\pi i} = \cos 2\pi + i \sin 2\pi = 1$$

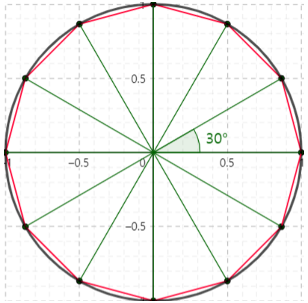
위를 이용하여 모든 자연수 n에 대해서 nth root of unity를 찾을 수 있습니다.

$$\omega = e^{\frac{2\pi}{n} i}$$

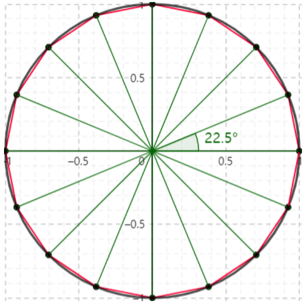
$$\omega^n = 1$$



n=7



n=12



n=16

Coefficient Representation to Point Representation

지금부터 n은 2의 거듭제곱 꼴이라고 가정하겠습니다. $n = 2^k, w = e^{\frac{2\pi}{n}i}$

길이가 n인 수열 {a} = [a_0, a_1, ⋯ , a_{n-1}]로 만든 다항식을 $f(x) = \sum_{i=0}^{n-1} a_i x^i$라고 합시다.

아래와 같은 총 n개의 서로 다른 점에서의 함숫값을 구하는 것이 목표입니다.

$$\underbrace{\{(\omega^0, f(\omega^0)), (\omega^1, f(\omega^1)), \cdots, (\omega^{n-1}, f(\omega^{n-1}))\}}_{\text{n points}}$$

Coefficient Representation to Point Representation

f(x)를 짝수차항과 홀수차항으로 분리해서 새로운 다항식을 아래처럼 만듭니다.

$$f_{even}(x) = a_0 + a_2 x + a_4 x^2 + \cdots + a_{n-2} x^{n/2-1}$$
$$f_{odd}(x) = a_1 + a_3 x + a_5 x^2 + \cdots + a_{n-1} x^{n/2-1}$$
$$f(x) = f_{even}(x^2) + x f_{odd}(x^2)$$

구하고자 했던 n개의 점들을 위 식에 대입해봅시다. 여기서 짚고 넘어가야 할 점은 $w^{n/2} = -1$이기 때문에 $w^{\frac{n}{2}+k} = -w^k$ $(\frac{n}{2} \le k < n)$이라는 것입니다.

$$f(\omega^k) = f_{even}(\omega^{2k}) + \omega^k f_{odd}(\omega^{2k}) \ (0 \le k < n/2)$$
$$f(\omega^k) = f_{even}(\omega^{2k}) - \omega^{k-n/2} f_{odd}(\omega^{2k}) \ (n/2 \le k < n)$$

Coefficient Representation to Point Representation

$$f(\omega^k) = f_{even}(\omega^{2k}) + \omega^k f_{odd}(\omega^{2k}) \ (0 \le k < n/2)$$
$$f(\omega^k) = f_{even}(\omega^{2k}) - \omega^{k-n/2} f_{odd}(\omega^{2k}) \ (n/2 \le k < n)$$

위 식을 이용해 $f(w^k)$는 $f_{even}(w^{2k}), f_{odd}(w^{2k})$를 알면 O(n)에 구할 수 있습니다.

w는 nth root of unity이기 때문에, w^2는 말하자면 $\frac{n}{2}$-th root of unity가 됩니다.
즉, n-1차 다항식 $f(x)$의 n개의 함숫값 $f(w^k)$들은 $\frac{n}{2} - 1$차 다항식 $f_{even}(x)$과 $f_{odd}(x)$의 $\frac{n}{2}$개의 함숫값 $f_{even}(w^{2k})$, $f_{odd}(w^{2k})$로부터 O(n)만에 얻을 수 있습니다.
심지어, w^2도 $\frac{n}{2}$-th root of unity입니다. 많이 본 모양새죠?

Coefficient Representation to Point Representation

n개의 점 $(w^k, f(w^k))$에 대해서 f(x)의 함숫값을 구하는 시간을 $T(n)$이라고 합시다.

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n)$$

따라서 총 시간복잡도는 $O(nlogn)$이 되며, 바로 이 과정이 FFT와 동일한 과정입니다.

$$\begin{aligned} X_k &= \sum_{n=0}^{N-1} x_n \cdot e^{-\frac{i2\pi}{N}kn} \\ &= \sum_{n=0}^{N-1} x_n \cdot \left[\cos\left(\frac{2\pi}{N}kn\right) - i \cdot \sin\left(\frac{2\pi}{N}kn\right)\right], \end{aligned} \quad \text{(Eq.1)}$$

Point Representation to Coefficient Representation

지금까지의 과정은 Coefficient Representation에서 Point Representation으로 다항식의 표현방법을 바꾼 것입니다.

이를 행렬로 표현하면 아래와 같습니다.

$$\begin{pmatrix} 1 & \omega^0 & (\omega^0)^2 & \cdots & (\omega^0)^{n-1} \\ 1 & \omega^1 & (\omega^1)^2 & \cdots & (\omega^1)^{n-1} \\ 1 & \omega^2 & (\omega^2)^2 & \cdots & (\omega^2)^{n-1} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & \omega^{n-1} & (\omega^{n-1})^2 & \cdots & (\omega^{n-1})^{n-1} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_{n-1} \end{pmatrix} = \begin{pmatrix} f(\omega^0) \\ f(\omega^1) \\ f(\omega^2) \\ \vdots \\ f(\omega^{n-1}) \end{pmatrix}$$

좌변의 행렬을 V라고 합시다.
그러면, Point Representation에서 Coefficient Representation으로 바꾼다는 것은 V의 역행렬 V^{-1}을 찾아서 양변에 곱해주는 것과 같습니다.

Point Representation to Coefficient Representation

다행히도, V의 역행렬은 알려져 있으며, (i,j)에 위치한 원소가 $\frac{w^{-ij}}{n}$인 행렬입니다.

그래서 그 역변환은 아래와 같이 행렬로 나타낼 수 있습니다.

$$\begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_{n-1} \end{pmatrix} = \frac{1}{n} \begin{pmatrix} 1 & \omega^0 & (\omega^0)^2 & \cdots & (\omega^0)^{n-1} \\ 1 & \omega^{-1} & (\omega^{-1})^2 & \cdots & (\omega^{-1})^{n-1} \\ 1 & \omega^{-2} & (\omega^{-2})^2 & \cdots & (\omega^{-2})^{n-1} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & \omega^{-(n-1)} & (\omega^{-(n-1)})^2 & \cdots & (\omega^{-(n-1)})^{n-1} \end{pmatrix} \begin{pmatrix} f(\omega^0) \\ f(\omega^1) \\ f(\omega^2) \\ \vdots \\ f(\omega^{n-1}) \end{pmatrix}$$

일단, w^{-1}도 nth root of unity임을 머릿속에 넣어두고 이 행렬을 다시 봅시다.

Point Representation to Coefficient Representation

$$\begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_{n-1} \end{pmatrix} = \frac{1}{n} \begin{pmatrix} 1 & \omega^0 & (\omega^0)^2 & \cdots & (\omega^0)^{n-1} \\ 1 & \omega^{-1} & (\omega^{-1})^2 & \cdots & (\omega^{-1})^{n-1} \\ 1 & \omega^{-2} & (\omega^{-2})^2 & \cdots & (\omega^{-2})^{n-1} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & \omega^{-(n-1)} & (\omega^{-(n-1)})^2 & \cdots & (\omega^{-(n-1)})^{n-1} \end{pmatrix} \begin{pmatrix} f(\omega^0) \\ f(\omega^1) \\ f(\omega^2) \\ \vdots \\ f(\omega^{n-1}) \end{pmatrix}$$

위 식과 coef->point의 행렬식과 비교를 w가 w^{-1}로 바뀌고 n으로 나눠준 꼴입니다.
즉, $f(w^k)$로 이루어진 다항식을 f'라고 하면 아래와 같은 점들 n개를 구하고 n으로 나눠준 것이 Coefficient Representation이 되는 것입니다.
이 과정을 Inverse FFT라고 부릅니다.

$$\underbrace{\{(\omega^0, f'(\omega^0)), (\omega^{-1}, f'(\omega^{-1})), \cdots, ((\omega^{-1})^{n-1}, f'((\omega^{-1})^{n-1}))\}}_{\text{n points}}$$

Polynomial Multicplication in O(nlogn)!

$$f(x) = a_0 + a_1x^1 + a_2x^2 + \ldots + a_{n-1}x^{n-1}$$

$$\updownarrow \quad O(nlogn)$$

$$(\omega_0, f(\omega_0)), (\omega_1, f(x_1)), (\omega_2, f(\omega_2)), \ldots, (\omega_{n-1}, f(\omega_{n-1}))$$

원래 목표로 했던 것은 두 다항식 f,g 의 곱이었습니다.

1. 각각 f, g의 Point Representation을 구합니다. O(nlogn)
2. 두 다항식의 곱 h의 Point Representation을 구합니다. O(n)
3. h의 Coefficient Representation을 구합니다. O(nlogn)

이것으로 O(nlogn)만에 다항식의 곱을 구하는 것이 가능해졌습니다.
따라서, 합성곱을 O(nlogn)만에 구하는 것이 가능해졌습니다.
그런데, n이 2의 거듭제곱 꼴이라는 조건이 있었죠?

Convolution in O(nlogn)

두 수열 a,b의 합성곱 c의 길이 |c|는 |a|+|b|-1이 됩니다.
|c|보다 크면서 가장 작은 2의 거듭제곱을 n이라고 정합니다.
아래와 같은 과정을 통해서 합성곱 c를 구할 수 있습니다.

1. 두 수열 a,b의 길이를 n으로 늘려줍니다. 모자란 부분은 0으로 채워넣습니다.
2. 두 수열을 계수로 가지는 n-1차 다항식 f, g를 만듭니다.
3. 각 다항식에 대해서 $w^0, w^1, w^2, \ldots, w^{n-1}$에 대한 함숫값을 FFT를 이용해서 구합니다.
4. 구한 함숫값들을 서로 곱해서 $h(w^k) = f(w^k)g(w^k)$를 구합니다.
5. h의 Coefficient Representation을 Inverse FFT를 이용해서 구합니다.
6. h(x)의 0부터 |a|+|b|-2차항 까지의 계수가 a,b의 합성곱 c가 됩니다.

Pseudocode

```
FFT(A, w):
    n <- sizeof A
    if n == 1: return
    split A into A_even, A_odd
    FFT(A_even, w*w)
    FFT(A_odd, w*w)
    p = 1
    for i...n/2
        A[i] <- A_even[i] + p * A_odd[i]
        A[i+n/2] <- A_even[i] - p * A_odd[i]
        p <- p * w
```

```
Convolution(a,b)
    n <- 1
    while n < |a|+|b|-1
        n <- n*2
    make size of a,b n by zero padding
    w <- exp(2*pi/n*i)
    a <- fft(a,w)
    b <- fft(b,w)
    c <- pointwise multiplication of a,b
    w <- exp(-2*pi/n*i)
    c <- fft(c,w)
    c <- c/n
    return c
```

Implementation

```cpp
#include<bits/stdc++.h>
using namespace std;
using cdbl = complex<double>;
const double PI = acos(-1);

void fft(vector<cdbl> &a, cdbl w) {
    int n = a.size();
    if(n == 1) return ;
    vector<cdbl> odd(n/2), even(n/2);
    for(int i=0;i<n;++i) {
        if(i%2) odd[i/2] = a[i];
        else even[i/2] = a[i];
    }
    fft(even,w*w);
    fft(odd,w*w);
    cdbl p(1,0);
    for(int j=0;j<n/2;++j) {
        a[j] = even[j] + p*odd[j];
        a[j+n/2] = even[j] - p*odd[j];
        p *= w;
    }
}
```

```cpp
vector<int> convolution(vector<int> &a, vector<int> &b) {
    vector<cdbl> f(a.begin(), a.end()), g(b.begin(), b.end());
    int n = 1;
    while(n < a.size() || n < b.size()) n <<= 1;
    n <<= 1;
    f.resize(n); g.resize(n);
    cdbl w(cos(2*PI/n),sin(2*PI/n));    // exp(2*pi/n)
    fft(a,w); fft(b,w);
    vector<cdbl> c(n);
    for(int i=0;i<n;++i) c[i] = a[i]*b[i];
    fft(c,cdbl(1,0)/w); // w^-1
    vector<int> ret(n);
    for(int i=0;i<n;++i) {
        c[i] = c[i] / cdbl(n,0);
         ret[i] = int(round(c[i].real()));
    }
    return ret;
}
```

https://bit.ly/3hPNn87

Performance Issue

FFT를 재귀적인 형태로 짜지 않고 비재귀로 짜는 것이 가능합니다!
여기서는 비재귀 구현체에 대한 설명 없이 넘어가겠습니다. 아래 링크의 재귀 풀기 부분을 보시면 자세히 나와있습니다.
https://casterian.net/archives/297

아래는 백준 15576번에서 재귀와 비재귀 속도차이입니다.

메모리, 시간 둘 다 차이가 심합니다. 팀노트에 꼭 비재귀 구현으로 넣어두도록 합시다.

18249956	seastar105	15576	맞았습니다!!	51904 KB	408 ms	C++14 / 수정
18100430	seastar105	15576	맞았습니다!!	84368 KB	876 ms	C++14 / 수정

“Any Question?”