Fast Fourier Transform: intro

Vladimir Smykalov

Toulouse, 2017

FFT introduction

Overall goal

Input: Given two polynomials $f(x) = a_0 + a_1 x + ... + a_n x^n$ and $g(x) = b_0 + b_1 x + ... + b_n x^n$

Output: Construct multiplication result $h(x) = f(x) \cdot g(x) = c_0 + c_1 x + ... + c_{2n} x^{2n}$

Straightforward solution -- $O(n^2)$

FFT approach -- $O(n \log n)$

Why do we even need to multiply polynomials

Applications are extremly wide!

Let's for example show the way to calculate for each number `x` from `1` to `n` the number of way to represent it as **sum of two primes** `x = p1 + p2`

Classic approach will only give us $O(n^2)$ complexity

Let's solve it with FFT!

Let $f(x) = (x^2 + x^3 + x^5 + x^7 + x^{11} + ...)$
Calculate $g(x) = f(x) \cdot f(x) = a_0 + a_1 x + a_2 x^2 + ...$

Coefficients $a_0, a_1, a_2, ...$ are the answer to the problem
(a_t is the number of ways to represent t as sum of two primes)

Algo time is just $O(n \log n)$!

How FFT works

Let $f(x)$ and $g(x)$ be polynomials with `degree f(x)*g(x) < N = 2^k`
Let $z^0, z^1, ..., z^{N-1}$ be complex roots of equation $x^N - 1 = 0$

To be more precise, let $z^t = \cos(\frac{2\pi t}{N}) + i\sin(\frac{2\pi t}{N})$

Then there are several steps:

1. Calculate values of $f(x)$ in points $z^0, z^1, \ldots$
2. Calculate values of $g(x)$ in points $z^0, z^1, \ldots$
3. Get values of $h(x)$ in points $z^0, z^1, \ldots$ using $h(z^t) = f(z^t) \cdot g(z^t)$
4. Interpolate this values back to coefficients of $h(x)$

The trick is to quickly do steps 1, 2, 4 in $O(N \log N)$ time

Let's start!

Have $f(x) = a_0 + a_1 x + ... + a_{N-1} x^{N-1}$ where $N = 2^k$
Want to calculate $f_t = f(z^t)$ for each t from `0` to `N-1`

Let's solve recursively!

Let $f(x) = f_{even}(x^2) + x f_{odd}(x^2)$

So $f_{even}(x) = a_0 + a_2 x + a_4 x^2 + ...$ and $f_{odd}(x) = a_1 + a_3 x + a_5 x^2 + ...$

Let's calculate recursively $f_{even}^{(t)} = f_{even}(z^{2t})$

Now, $f(z^t) = f_{even}(z^{2t}) + z^t f_{odd}(z^{2t})$
$f_t = f(z^t) = f_{even}^{(t)} + z^t f_{odd}^{(t)}$ for each `0 <= t < N/2`
$f_t = f(z^t) = f_{even}^{(t-N/2)} + z^t f_{odd}^{(t-N/2)}$ for each `N/2 <= t < N`

Awesome!

Let's write pseudocode

```
def fft(a, N): # computes values of polynomial (sum a_i * x^i) in roots of x^N - 1 = 0
    if N == 1:
        return [a[0]]

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

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

    # reconstruct f values
    for i in 0 .. N/2-1:
        f[i] =     f_even[i] + z[i] * f_odd[i]
        f[i+N/2] = f_even[i] + z[i+N/2] * f_odd[i]

    return f
```

How fast is algo?

Similar to segment trees, overall complexity is $O(N \log N)$

$$T(1) = 1$$
$$T(N) = 2\,T(N/2) + N$$

Solution:

$$T(N) = N \log_2(2N)$$

Let's remember steps

1. Calculate values of $f(x)$ in points $z^0, z^1, \ldots$
2. Calculate values of $g(x)$ in points $z^0, z^1, \ldots$
3. Get values of $h(x)$ in points $z^0, z^1, \ldots$ using $h(z^t) = f(z^t) \cdot g(z^t)$
4. Interpolate this values back to coefficients of $h(x)$

Now we now how to do steps 1 and 2

But how to run interpolation?

Magic: reverse and run `fft`

Now we have $f_t = f(z^t)$

Want to get back $a_0, a_1, ...$ from $f(x) = a_0 + a_1 x + ...$

... but how?

Let's reverse segment `[1, N-1]` and write it as polynom

Let's $F(x) = f_0 + f_{N-1} x + f_{N-2} x^2 + ... + f_1 x^{N-1}$

Let's run `fft(F, N)`

Now, $a_t = \frac{1}{N} F(z^t)$

Magic (explanation)

What is $F(z^t)$?

$$F(z^t) = f_0 + f_{N-1}z^t + f_{N-2}z^{2t} + ... + f_1 z^{(N-1)t} =$$

$$= f(z^0) + f(z^{N-1})z^t + f(z^{N-2})z^{2t} + ... + f(z^1)z^{(N-1)t} =$$

$$= (a_0 + a_1 z^0 + ...) + (a_0 + a_1 z^{N-1} + ...)z^t + (a_0 + a_1 z^{N-2} + ...)z^{2t} + ... =$$

$$= a_0(z^0 + z^t + z^{2t} + ...) + a_1(z^0 + z^{(N-1)+t} + z^{(N-2)+2t} + ...) + ... =$$

$$= \sum a_s(z^0 + z^{(t-s)} + z^{2(t-s)} + ... + z^{(N-1)(t-s)})$$

So much math... wait!

$z^0 + z^{(t-s)} + z^{2(t-s)} + ... + z^{(N-1)(t-s)}$ is `0` if `t != s`
$z^0 + z^{(t-s)} + z^{2(t-s)} + ... + z^{(N-1)(t-s)}$ is `N` if `t == s`

That means $F(z^t) = N \cdot a_t$!

Time to finish code for polynom multiplication!

```
def mult(a, b): # multplies {a} and {b} polynoms and returns result {c}
    # Step 1 and 2
    f = fft(a, N)
    g = fft(b, N)

    # Step 3
    for i in 0 .. N-1:
        h[i] = f[i] * g[i]

    # Step 4
    reverse h[1 .. N-1]
    c = fft(h, N)

    # finishing touches
    for i in 0 .. N-1:
        c[i] = c[i] / N

    return c
```

quick fix, let's make code shorter

```
def mult(a, b): # multplies {a} and {b} polynoms and returns result {c}
    # Step 1 and 2
    f = fft(a, N)
    g = fft(b, N)

    # Step 3
    for i in 0 .. N-1:
        h[i] = f[i] * g[i] / N

    # Step 4
    reverse h[1 .. N-1]
    c = fft(h, N)

    return c
```

Implementation

Notes

- solve fft-problems in `c++` and only `c++`
- there is `std` class for complex numbers `complex<double>`
- try to write as efficient as possible (for example, creating new `vector`s inside recursion for `f_odd` and `f_even` is **very** timeconsumable, better to reuse memory)
- use your head to write code and not .... other things

How to write FFT efficiently as a whole another lecture!

But this knowledge is enough to solve basic problems!