Report 2 FFT Multiplication - 1930026143

Question recall and the solution

1. Before starting doing the project, we should comb through the leading knowledge. One I want to mention that is Discrete Fourier Transform (DFT). And the definition of it: let $\boldsymbol{\omega}$ be a primitive n^{th} root of unity, then the DFT is represented as:

$$X(k) = DFT(x(t)) = \sum_{t=0}^{n-1} x(t)\omega^{tk}$$

$$x(t) = IDFT(X(k)) = \frac{1}{n}\sum_{k=0}^{n-1} X(k)\omega^{-tk}$$

And it does not base on the plural system which need to be combined with the Euler's formula. And the NTT also like the DFT just with a litter change. We can also use a matrix to represent the process of Discrete Fourier Transform:

$$\begin{bmatrix} f_0 \\ f_1 \\ f_2 \\ \vdots \\ f_{N-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & \overline{W} & \overline{W}^2 & \cdots & \overline{W}^{N-1} \\ 1 & \overline{W}^2 & \overline{W}^4 & \cdots & \overline{W}^{2(N-1)} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & \overline{W}^{N-1} & \overline{W}^{2(N-1)} & \cdots & \overline{W}^{(N-1)^2} \end{bmatrix} \begin{bmatrix} C_0 \\ C_1 \\ C_2 \\ \vdots \\ C_{N-1} \end{bmatrix}$$

2. Euler's Totient Function $\emptyset(n)$ which is defined as the number of positive integers less than n and relatively prime to n. And the algorithm of it: $\emptyset(1) = 1$; For a prime number p, $\emptyset(p) = p - 1$. If it is not a prime, it will be the numbers of smaller integers are not divisible.
And it has some important and useful property:
   - For every a and n that are relatively prime:
$$a^{\emptyset(n)} \ (mod\ n) \ = \ 1 \ (mod\ n)$$
   - An alternative form is:
$$a\hat{}\emptyset(n) + 1 \ (mod\ n) \ = \ a \ (mod\ n)$$

a is a primitive n^{th} root of unity modulo p if and only if $a\hat{}n \ (mod\ p) \ = \ 1 \ (mod\ p)$ and $a^k \ (mod\ p) \ \neq \ 1 \ (mod\ p)$ for any positive integer $k < n$. The number a is also called the generator of the ring of integers modulo p (finite ring Zp). According to the property, we can introduce to the Number Theoretic Transform (NTT) and INNT (inverse):

$$X(k) = NTT\big(x(t)\big) = \sum_{t=0}^{n-1} x(t)\omega^{tk} \ mod\ q$$

$$x(t) = INTT\big(X(k)\big) = n^{-1}\sum_{k=0}^{n-1} X(k)\omega^{-tk} \ mod\ q$$

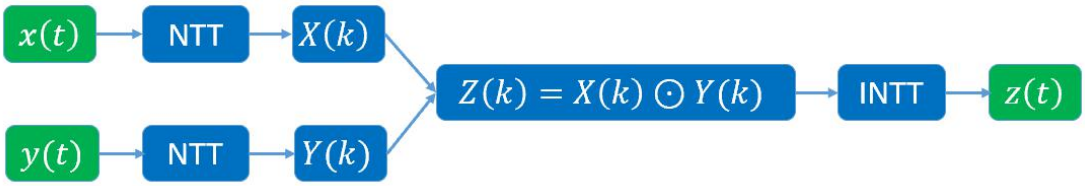
Then in order to accurate the speed up the algorithm when we do multiplication by using NTT, we can use cyclic convolution $z(i) = CyclicConv\big(x(t), y(t)\big)_i = \sum_{j=0}^{n-1} x\big((i-j)\ mod\ n\big)\cdot$

$y(j)$ instead of the linear convolution $z(i) = LinearConv\big(x(t), y(t)\big)_i = \sum_{j=0}^{s-1} x(i-j) \cdot y(j)$ whose' result is equivalent to each other. That's since we can omit a lot of operations with the modulo, which means that there are duplicates around a "period" can be avoid computing repeatedly.

3. The core of algorithm thinking is using cyclic convolution to multiply the Number Theoretic Transform of the two vectors which should be inverse to the time domain function by INTT. Then we get the result of the multiplication.

$$z(i) = CyclicConv\big(x(t), y(t)\big)_i = \sum_{j=0}^{n-1} x\big((i-j)\ mod\ n\big) \cdot y(j)$$

$$= INTT\Big(NTT\big(x(t)\big) \odot NTT\big(y(t)\big)\Big)_i = INTT\big(X(k) \odot Y(k)\big)_i$$



4. Then we can use code to solve the project:
   a. The first thing we should do is zero padding. If the input vector length s is not power of 2, zeroes are needed to be padded on the higher indexes of the vector to make sure that the vector length $s = 2^i$:

```
def zero_padding(vec):
    #---------------------------------
    # please provide your code here
    vec_len = len(vec)
    cnt= 0
    while(pow(2,cnt)<=vec_len):
        cnt+=1
    for i in range(pow(2, cnt+1)-vec_len):
        vec.append(0)
    #---------------------------------
    return vec
```

   b. In order to avoid overflow, we need to define the minimum working modulus. By the limitation and Dirichlet's theorem, it guarantees that such a prime number that convolution on two vectors of length n where each input coefficient value is at most m must exist.

```
def find_modulus(veclen, minmod):
    check_int(veclen)
    check_int(minmod)
    if veclen < 1 or minmod < 1:
        raise ValueError()
    #---------------------------------
    # please provide your codes here
    c = 1
    n = c*veclen + 1
    while n<minmod or not is_prime(n):
        c+=1
        n = c*veclen + 1
    #---------------------------------
    return n
```

c. Select some appropriate integer $c \geq 1$ and define $q = cn + 1$ as the modulus such that $q \geq M$, and q to be a prime number. The selection of generator is divided into two functions: $find\_generator$ which use loop for each digit of vector to call the $is\_generator$ and $is\_generator$ use the judgement statement to determent if the integer is generator. Totient must equal the Euler phi function of mod. If mod is prime, an answer must exist.

```
def is_generator(val, totient, mod):
    check_int(val)
    check_int(totient)
    check_int(mod)
    if not (0 <= val < mod):
        raise ValueError()
    if not (1 <= totient < mod):
        raise ValueError()
    #---------------------------------
    # please provide your codes here
    prime_factors = unique_prime_factors(totient)
    det = True
    for i in prime_factors:
        if val**(totient//i) % mod == 1:
            det = False
            break
    #---------------------------------
    return det
```

d. When we get the generator, modulus and the appropriate integer, we can get the root of the NNT ω. The function will returns an arbitrary primitive n^{th} root of unity modulo mod.

```
def find_primitive_root(degree, totient, mod):
    print ('degree, totient, mod',degree, totient, mod)
    check_int(degree)
    check_int(totient)
    check_int(mod)
    if not (1 <= degree <= totient < mod):
        raise ValueError()
    if totient % degree != 0:
        raise ValueError()
    #---------------------------------
    # please provide your codes here
    # totient = mod-1
    g = find_generator(totient, mod)
    c = totient // degree
    root = pow(g, c) % mod
    #---------------------------------
    return root
```

e. Take the transformation of the two vectors that you want to multiply and compute the point-wise multiplication of the two vectors modulo.

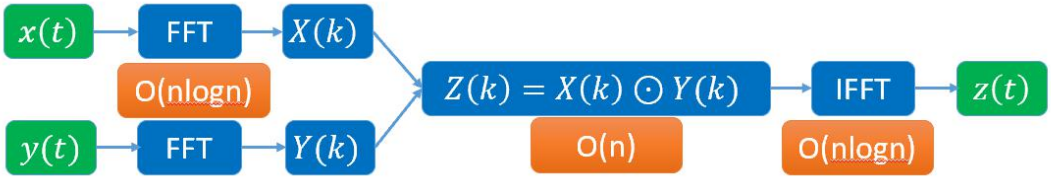
```
    for k in range(len(invec)):
        res = 0
        for t in range(len(invec)):
            res += invec[t]* pow(root, t*k)
        X_k = res % mod
        outvec.append(X_k)
```

```
#point-wise multiplication in frequency domain
vec2 = [(x * y % mod) for (x, y) in zip(vec0, vec1)]
```

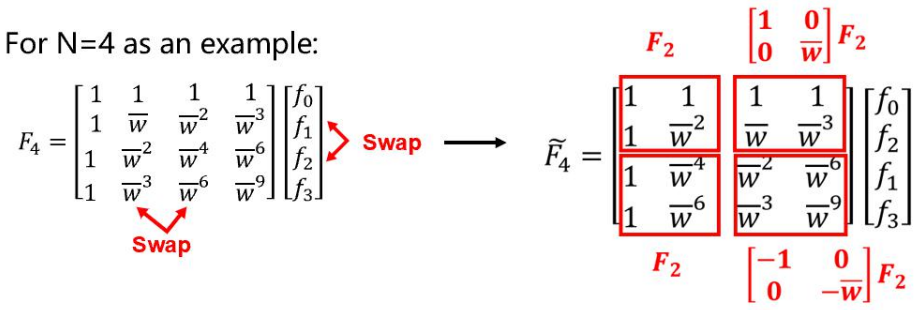
f. Compute the inverse NTT of the result in step e: INTT(X(k) ⊙ Y(k) mod q) . In this case, **remember to transform the result as integer or will get some error.**

```
def idft_fft(invec, root, mod):
    outvec = dft_fft(invec, reciprocal(root, mod), mod)
    scaler = reciprocal(len(invec), mod)
    return [int(val * scaler % mod) for val in outvec]
```

5. Then we keep on analyzing the Fast Fourier Transform (FFT) Structure on NTT which can help us to speed up the multiplication by NTT. We can find that it just add the FFT before the multiplication of two NNT vectors.



It depends on the principle that the DFT of an arbitrary composite size $n = n_1 n_2$ in terms of n_1 smaller DFTs of sizes n_2, recursively. We may understand that by the operation of matrix. Here we use $N = 4$ as example:



We use some linear algebra skills to solve this. Initially, we should change the order of the vector by bit reverse. For n rows, there are n square multiplications and n square minus n additions are required. So we should reduce cost by the time of inner product. Swap even and odd columns and the odd and even rows for the f vector. Then it can be simplified and be **generalization, every n to the power of two matrix can resolve into the lower.**

```
    def bit_reverse(x, bits):
        y = 0
        for i in range(bits):
            y = (y<<1) | (x&1)
            x >>= 1
        return y

    lgn = math.log(n,2)
    x = [vector[i] for i in [bit_reverse(i, levels) for i in range(n)]]
    X = n * [0]
    for j in range(int(lgn)):
        for k in range(int(n/2)):
            p = math.floor(k / pow(2, (lgn-1-j))) * pow(2, lgn-1-j)
            X[k] = (x[2*k] + x[2*k + 1] * pow(root, p)) % mod
            X[k+int(n/2)] = (x[2*k] - x[2*k + 1] * pow(root, p)) % mod
        if j != lgn - 1:
            for k in range(n):
                x[k] = X[k]
    vector = X
```

We divide odd and even for the image shape and keep recursion to divide, when we get the

basic case and then just calculate the result by NTT.

All the dft_ntt and $idft\_ntt$ should be changed to dft_fft and $idft\_fft$:

```
def fft_mult(in0, in1):

    #transform integer to vector
    base = 10
    vec0 = int2vec(in0,base)
    vec1 = int2vec(in1,base)

    #input validation check
    if not (0 < len(vec0) == len(vec1)):
        raise ValueError()
    if any((val < 0) for val in itertools.chain(vec0, vec1)):
        raise ValueError()

    #zero padding the vectors to length of power of 2
    vec0 = zero_padding(vec0);
    vec1 = zero_padding(vec1);

    #parameter selction
    n = len(vec0)
    minmod = find_minmod(vec0, vec1)
    mod = find_modulus(n, minmod)
    root = find_primitive_root(n, mod-1, mod)

    #forward transforms
    vec0 = dft_fft(vec0, root, mod)
    vec1 = dft_fft(vec1, root, mod)

    #point-wise multiplication in frequency domain
    vec2 = [int(x * y % mod) for (x, y) in zip(vec0, vec1)]

    #inverse transform
    result = idft_fft(vec2, root, mod)

    #transform vector to integer
    product = 0
    for i in range(len(result)):
        product += result[i] * base**i
    return product
```

Then we can briefly analyze their complexity. When the length of vector is N, NTT needs to compute approximately N^2 multiplications and N^2 additions. Decompose the DFT of N points into two NTT of $\frac{N}{2}$ points, so that the total calculation of two NTT of $\frac{N}{2}$ points is only half of the original, that is, $\left(\frac{N}{2}\right)^2 + \left(\frac{N}{2}\right)^2 = \frac{N^2}{2}$. In this way, the decomposition can continue, and then decompose $\frac{N}{2}$ into NTT of $\frac{N}{4}$ points, etc. The NTT at $N = 2x$ can be decomposed into the NTT at 2 points, so that the computation can be reduced to $\left(\frac{N}{2}\right) * log2N$ multiplications and $N * log2N$ addition.