Notes for NTT, version 0.2

Antti Roeyskoe

31.5.2019

1 Number-Theoretic Transform

1.1 Goal

Given a prime p and integer d such that $d = 2^k$ for some k and $2^{k+1} \mid p-1$, and two polynomials P and Q

$$P(x) = \sum_{i=0}^{d-1} a_i x^i$$
$$Q(x) = \sum_{i=0}^{d-1} b_i x^i$$

The algorithm finds the remainders mod p of the coefficients c_i of the product polynomial $R(x) = P(x)Q(x)$:

$$\begin{aligned} R(x) &= \sum_{i=0}^{2d-1} c_i x^i \\ &= \sum_{i=0}^{2d-1} \left(\sum_{j=0}^{i} a_j b_{i-j} \right) x^i \\ &= P(x)Q(x) \end{aligned}$$

where a_j and b_j are zero for terms that don't exist ($j < 0$ or $d \leqslant j$)

1.2 Intuition

Define vectors

$$a = [a_0, \ldots, a_{d-1}, 0, \ldots, 0]$$
$$b = [b_0, \ldots, b_{d-1}, 0, \ldots, 0]$$
$$c = [c_0, \ldots, c_{2d-1}]$$

of length $2d$.
Let x be some integer. Let M be a $2d \times 2d$ matrix where $M_{i,j} = x^{ij}$ (zero-indexed).
Now

$$aM = [P(x^0), \ldots, P(x^{2d-1})]$$
$$bM = [Q(x^0), \ldots, Q(x^{2d-1})]$$

(proof in appendix) and

$$\begin{aligned} cM &= [R(x^0), \ldots, R(x^{2d-1})] \\ &= [P(x^0)Q(x^0), \ldots, P(x^{2d-1})Q(x^{2d-1})] \\ &= aM \circ bM \end{aligned}$$

Where $\circ$ is the elementwise matrix product.
We will choose x such that M^{-1} exists, and then get

$$\begin{aligned} c &= cMM^{-1} \\ &= (aM \circ bM)M^{-1} \end{aligned}$$

So if we can calculate vM and vM^{-1} for a vector v in time $O(n \log n)$, we can calculate c from a and b in $O(n \log n)$. It turns out that this is possible! But first we need some number-theoretic background.

1.3 Number-Theoretic Background

Let p be a prime. Let P to denote the group $\mathbb{Z}/p\mathbb{Z}$ of integers mod p.

Definition 1.1. *The order $ord(g)$ of an element $g \in P$ is the minimum positive integer such that $g^{ord(g)} \equiv 1$ mod p.*

Definition 1.2. *An element $g \in P$ a generator if $ord(g) = p - 1$*

Let $g \in P, g \not\equiv 0$ be some nonzero element mod p. The following statements are true:

- $g^{p-1} \equiv 1 \bmod p$ (fermat's little theorem)
- If $g^a \equiv 1 \bmod p$, then $ord(g) \mid a$.
- If $b \mid ord(g)$, then $ord(g^b) = \frac{ord(g)}{b}$.
- For odd primes, exactly $\frac{p-1}{2}$ elements in P are generators.
- If g is a generator of p, then $\frac{1}{p}$ is also a generator of p.
- If $ord(g) = 2$, $g \equiv -1 \bmod p$.

(proof in the appendix)

1.4 Algorithm

Recall that we require $d = 2^k$ for some integer k, and that $2^{k+1} \mid p - 1$.
Let $g \in P$ be a generator. For any $n \mid 2^k$, set $x_n \equiv g^{\frac{p-1}{n}} \bmod p$, and let $M[n]$ be a $n \times n$ matrix where

$$M[n]_{i,j} \equiv x_n^{ij} \bmod p$$

Note that this $M[n]$ is of the wanted type with $x = x_n$. It turns out that $M[n]^{-1}$ exists, and that we can calculate $vM[n]$ fast. Next we'll show how:

1.4.1 Forward Direction

Let n be some power of two such that $n \mid 2^k$. Let $v = [v_0, \ldots, v_{n-1}]$, $v_i \in P$ be a vector with length n. Define $f(v) : v \mapsto vM[n]$. We'll now show how to compute $f(v)$ in time $O(n \log n)$.
If $n = 1$, then since $x_1 \equiv 1$, $vM[n] = [v_0]$. When $n > 1$, define

$$\begin{aligned} even(v) &= [v_0, v_2, \ldots, v_{n-2}] \\ odd(v) &= [v_1, v_3, \ldots, v_{n-1}] \end{aligned}$$

(note that n is a power of two greater than 1, so it is even.) Let $0 \leqslant j < n$ be some index, and define $h = \frac{n}{2}$. we have

$$\begin{aligned} (vM[n])_j &\equiv \sum_{i=0}^{n-1} v_i M[n]_{i,j} \\ &\equiv \sum_{i=0}^{n-1} v_i x_n^{ij} \\ &\equiv \sum_{i=0}^{h-1} v_{2i} x_n^{2ij} + \sum_{i=0}^{h-1} v_{2i+1} x_n^{(2i+1)j} \\ &\equiv \sum_{i=0}^{h-1} v_{2i} \left(x_n^2\right)^{ij} + x_n^j \sum_{i=0}^{h-1} v_{2i+1} \left(x_n^2\right)^{ij} \mod p \end{aligned}$$

We have $x_n^2 = x_h$ by definition:

$$\begin{aligned} x_n^2 &\equiv \left(g^{\frac{p-1}{n}}\right)^2 \\ &\equiv g^{2\frac{p-1}{n}} \\ &\equiv g^{\frac{p-1}{h}} \\ &\equiv x_h \mod p \end{aligned}$$

therefore

$$\begin{aligned} (vM[n])_j &\equiv \sum_{i=0}^{h-1} v_{2i} \left(x_n^2\right)^{ij} + x_n^j \sum_{i=0}^{h-1} v_{2i+1} \left(x_n^2\right)^{ij} \\ &\equiv \sum_{i=0}^{h-1} v_{2i} x_h^{ij} + x_n^j \sum_{i=0}^{h-1} v_{2i+1} x_h^{ij} \mod p \end{aligned}$$

If $j < h$, then

$$\begin{aligned}(vM[n])_j &\equiv \sum_{i=0}^{h-1} v_{2i} x_h^{ij} + x_n^j \sum_{i=0}^{h-1} v_{2i+1} x_h^{ij} \\ &\equiv f(even(v))_j + x_n^j f(odd(v))_j \bmod p\end{aligned}$$

If $j \geqslant h$, We have

$$1 \equiv 1^i \equiv x_1^i \equiv \left(x_h^h\right)^i \equiv x_h^{ih} \bmod p$$

Set $j^{'} = j - h$. Now $0 \leqslant j^{'} < h$, so

$$\begin{aligned}(vM[n])_j &\equiv (vM[n])_{j^{'}+h} \\ &\equiv \sum_{i=0}^{h-1} v_{2i} x_h^{i(j^{'}+h)} + x_n^{j^{'}+h} \sum_{i=0}^{h-1} v_{2i+1} x_h^{i(j^{'}+h)} \\ &\equiv \sum_{i=0}^{h-1} v_{2i} x_h^{ij^{'}} x_h^{ih} + x_n^{j^{'}+h} \sum_{i=0}^{h-1} v_{2i+1} x_h^{ij^{'}} x_h^{ih} \\ &\equiv \sum_{i=0}^{h-1} v_{2i} x_h^{ij^{'}} + x_n^{j^{'}+h} \sum_{i=0}^{h-1} v_{2i+1} x_h^{ij^{'}} \\ &\equiv f(even(v))_{j^{'}} + x_n^{j^{'}+h} f(odd(v))_{j^{'}} \\ &\equiv f(even(v))_{j^{'}} + x_n^h x_n^{j^{'}} f(odd(v))_{j^{'}} \\ &\equiv f(even(v))_{j^{'}} - x_n^{j^{'}} f(odd(v))_{j^{'}} \bmod p\end{aligned}$$

Since $ord\left(x_n^h\right) = 2$, and therefore $x_n^h \equiv -1 \bmod p$.
Therefore for $0 \leqslant j < h$ we have:

$$\begin{aligned}(vM[n])_j &\equiv f(even(v))_j + x_n^j f(odd(v))_j \\ (vM[n])_{j+h} &\equiv f(even(v))_j - x_n^j f(odd(v))_j\end{aligned}$$

So when we have $f(even(v))$ and $f(odd(v))$, we can easily calculate $f(v)$ in linear time. Since $even(v)$ and $odd(v)$ have size $h = \frac{n}{2}$, we can calculate them recursively. This gives a $O(nlogn)$ algorithm.

1.4.2 Reverse Direction

To find $M[n]^{-1}$, note that we only used the fact that g is a generator. But $\frac{1}{g}$ is also a generator. Set $g^{'} = \frac{1}{g}$, and define $M^{'}[n]$ similarly as how $M[n]$ is defined, except that it uses $g^{'}$ instead of g. We have

$$M[n]M^{'}[n] = nI[n]$$

(proof in the appendix) Where $I[n]$ is the identity matrix of size $n \times n$. Therefore $\frac{1}{n}M^{'}[n]$ is the inverse matrix of $M[n]$. Furthermore, we have

$$v\left(\frac{1}{n}M^{'}[n]\right) = \left(v\frac{1}{n}\right)M^{'}[n]$$

So we can multiply a vector with $\frac{1}{n}M^{'}[n]$ the same way as we multiplied it with $M[n]$, just by changing the generator we give to the function.

2 Code and Improvements

2.1 Recursive Code

All codes will have the same includes and definitions. Here we define the prime and generator we will be using.

```
#include <iostream>
#include <vector>
using namespace std;
using ll = long long;
const int P = 998244353; // 2^21 | P-1
const int G = 3; // 3 is a generator of P
```

The main NTT-function. It modifies the input vector instead of building a new one.

```
void ntt(vector<int>& v, int x_n) {
        int h = v.size()/2;
        vector<int> even(h);
        vector<int> odd(h);
        for (int i = 0; i < h; ++i) {
                even[i] = v[2*i];
                odd[i] = v[2*i+1];
        }

        if (h > 1) {
                int x_h = (ll)x_n*x_n % P;
                ntt(even, x_h);
                ntt(odd, x_h);
        }

        ll mult = 1; // (x_n)^i
        for (int i = 0; i < h; ++i) {
                v[i] = (even[i] + mult * odd[i]) % P;
                v[i+h] = (even[i] - mult * odd[i]) % P;
                if (v[i+h] < 0) v[i+h] += P;
                mult = mult*x_n % P;
        }
}
```

Here we have the usual function for calculating $a^b \bmod P$, and a helper function wrapping the calls to NTT made when multiplying two polynomials a and b. If vectors a and b contain the coefficients $a[i] = a_i$, $b[i] = b_i$ of polynomials A and B, then the result vector c will contain the coefficients $c[i] = c_i$ of $C = AB$.

```
ll modPow(ll a, ll b) {
        if (b & 1) return a * modPow(a, b-1) % P;
        if (b == 0) return 1;
        return modPow(a*a % P, b / 2);
}

vector<int> polyMult(const vector<int>& a, const vector<int>& b) {
        int as = a.size();
        int bs = b.size();
        int n = 1;
        while(n < (as + bs)) n <<= 1;
        int x_n = modPow(G, (P-1)/n);
        int inv_x_n = modPow(x_n, P-2);
        int inv_n = modPow(n, P-2);

        vector<int> ap (n, 0);
        vector<int> bp (n, 0);
        for (int i = 0; i < as; ++i) ap[i] = a[i] % P;
        for (int i = 0; i < bs; ++i) bp[i] = b[i] % P;

        ntt(ap, x_n);
        ntt(bp, x_n);

        vector<int> cp(n);
        for (int i = 0; i < n; ++i) {
                ll prod = (ll)ap[i] * bp[i] % P;
                cp[i] = prod * inv_n % P;
        }

        ntt(cp, inv_x_n);

        cp.resize(as + bs - 1);
        return cp;
}
```

2.2 Iterative Code

TODO

3 Tricks with NTT

3.1 Number of Paths

A common problem is that we want to calculate the number of paths from some source cell (x_s, y_s) to other cells in a grid, such that at every step we can move from (x, y) to $(x+1, y)$ or $(x, y+1)$, and there are some blocked cells we cannot enter.

Given a square with side length n containing no blocked cells, and how many ways exist to get from the source to each cell on the left side or on the bottom side of the square, we want to calculate the number of ways to get from the source to cells on the top side and right side of the square.

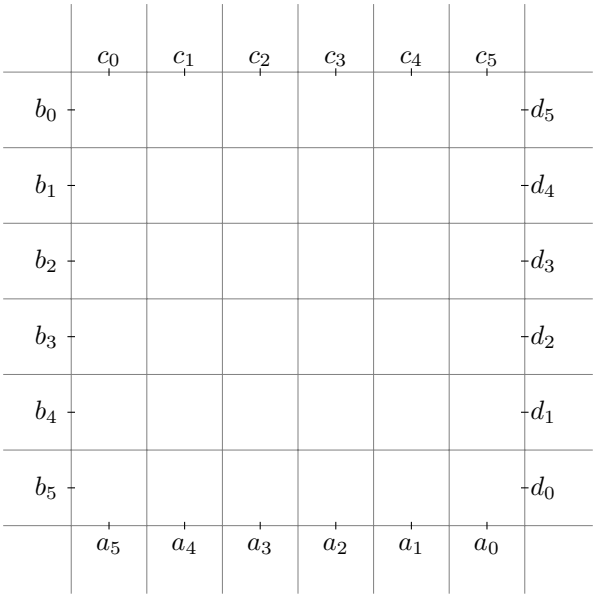

Figure 1: Indexing of the arrays. The arrays a, b are the input arrays, and c, d the output arrays.

This is possible since any path from the source cell to a cell in the square must travel through its left or bottom side. To calculate the number of ways to reach the top and right sides, we can use simple $O(n^2)$ dp, setting $ways[x][y] = (ways[x-1][y] + ways[x][y-1])\%P$.
If P is a suitable prime for NTT, this can be done faster, in time $O(n \log n)$.
Let $c1[j]$ be the vector representing the number of ways to get from the source to c_j through some node in a, and $c2[j]$ respectively be the number of ways to get from the source to c_j through some node in b. Note that every path is counted exactly once among these two, so $c[j] = c1[j] + c2[j]$.
We have

$$c1[j] = \sum_{i=n-1-j}^{n-1} a[i] \binom{i+j}{n-1}$$

$$c2[j] = \sum_{i=0}^{n-1} b[i] \binom{i+j}{j}$$

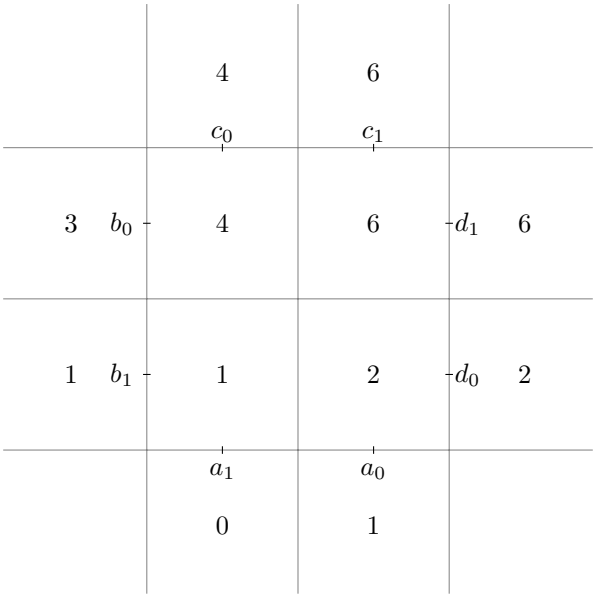

Figure 2: Example values and ways to reach the positions. Note that $c_{n-1} = d_{n-1}$, since they both represent the number of ways to reach the up-right corner.

So these can be calculated with the following matrix multiplications:

$$aM1 = a \begin{bmatrix} 0 & \dots & 0 & \dots & \binom{n-1}{n-1} \\ \vdots & & \vdots & & \vdots \\ 0 & \dots & \binom{i+j}{n-1} & \dots & \binom{n-1+i}{n-1} \\ \vdots & & \vdots & & \vdots \\ \binom{n-1}{n-1} & \dots & \binom{n-1+j}{n-1} & \dots & \binom{2n-2}{n-1} \end{bmatrix} = c1$$

$$bM2 = b \begin{bmatrix} \binom{0}{0} & \dots & \binom{j}{j} & \dots & \binom{n-1}{n-1} \\ \vdots & & \vdots & & \vdots \\ \binom{i}{0} & \dots & \binom{i+j}{j} & \dots & \binom{n-1+i}{n-1} \\ \vdots & & \vdots & & \vdots \\ \binom{n-1}{0} & \dots & \binom{n-1+j}{j} & \dots & \binom{2n-2}{n-1} \end{bmatrix} = c2$$

Changing the second to factorials we get

$$M2 = \begin{bmatrix} \frac{0!}{0!0!} & \cdots & \frac{j!}{j!0!} & \cdots & \frac{(n-1)!}{(n-1)!0!} \\ \vdots & & \vdots & & \vdots \\ \frac{i!}{0!i!} & \cdots & \frac{(i+j)!}{j!i!} & \cdots & \frac{(n-1+i)!}{(n-1)!i!} \\ \vdots & & \vdots & & \vdots \\ \frac{(n-1)!}{0!(n-1)!} & \cdots & \frac{(n-1+j)!}{j!(n-1)!} & \cdots & \binom{(2n-2)!}{(n-1)!(n-1)!} \end{bmatrix}$$

$$= diag\left(\begin{bmatrix} \frac{1}{0!} \\ \vdots \\ \frac{1}{i!} \\ \vdots \\ \frac{1}{(n-1)!} \end{bmatrix}\right) M3 diag\left(\begin{bmatrix} \frac{1}{0!} \\ \vdots \\ \frac{1}{i!} \\ \vdots \\ \frac{1}{(n-1)!} \end{bmatrix}\right)$$

Where *diag* makes a diagonal matrix with the elements of the vector on the diagonal, and

$$M3 = \begin{bmatrix} 0! & \cdots & j! & \cdots & (n-1)! \\ \vdots & & \vdots & & \vdots \\ i! & \cdots & (i+j)! & \cdots & (i+n-1)! \\ \vdots & & \vdots & & \vdots \\ (n-1)! & \cdots & (n-1+j)! & \cdots & (2n-2)! \end{bmatrix}$$

Note that we can multiply a vector with a diagonal matrix in $O(n)$, and that both $M1$ and $M3$ have the form

$$\begin{bmatrix} x_0 & \cdots & x_j & \cdots & x_{n-1} \\ \vdots & & \vdots & & \vdots \\ x_i & \cdots & x_{i+j} & \cdots & x_{i+n-1} \\ \vdots & & \vdots & & \vdots \\ x_{n-1} & \cdots & x_{n-1+j} & \cdots & x_{2n-2} \end{bmatrix}$$

Matrices in this form are called Hankel matrices. We'll later show how to multiply a vector with a matrix in this form in $O(n \log n)$ time using NTT.
So now we just calculate $c = c1 + c2$ in $O(n \log n)$ time. d is symmetric, swapping a and b. We just need to calculate the values $\binom{k}{n-1}$ and $k!$ for $0 \leqslant k < 2n-1$. But $\binom{k}{n-1} = \frac{k!}{(n-1)!k!}$, so it is enough to precalculate the values and modular inverses of $0 \leqslant k < 2n-1$. This can be done in $O(n \log n)$ time by calculating inverses with the extended euclidean algorithm, assuming we can do arithmetic operations on numbers up to P in $O(1)$ time.

3.1.1 Multiplying vectors with Hankel matrices

We want to multiply a vector $v = [v_0, \ldots, v_{n-1}]$ with the Hankel matrix

$$M = \begin{bmatrix} x_0 & \ldots & x_j & \ldots & x_{n-1} \\ \vdots & & \vdots & & \vdots \\ x_i & \ldots & x_{i+j} & \ldots & x_{i+n-1} \\ \vdots & & \vdots & & \vdots \\ x_{n-1} & \ldots & x_{n-1+j} & \ldots & x_{2n-2} \end{bmatrix}$$

To do this, we use NTT on the vectors $v^{'} = [v_0, \ldots, v_{n-1}, 0, \ldots, 0]$, $x^{'} = [x_{2n-2}, \ldots, x_0]$ of length $2n-1$. For $0 \leqslant i < n$, the $2n-2-i$th entry of the result will be:

$$\begin{aligned} NTT(v^{'}, x^{'})_{2n-2-i} &= \sum_{j=0}^{2n-2-i} v^{'}_j x^{'}_{2n-2-i-j} \\ &= \sum_{j=0}^{n-1} v_j x^{'}_{2n-2-(i+j)} \\ &= \sum_{j=0}^{n-1} v_j x_{i+j} \\ &= (vM)_i \end{aligned}$$

So we can set $res_i = NTT(v^{'}, x^{'})_{2n-2-i}$ for $0 \leqslant i < n$

4 appendix

TODO