Application of Number Theoretic Transform and Binary Exponentiation for Generalizing Sum of Periodic Binomial Combination

Farhan Nafis Rayhan - 13522037
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail (gmail): farhannafis281004@gmail.com

***Abstract*—In this paper, I proposed a solution to generalize the sum of periodic binomial combination. Inspired by a classical math olympiad problem, a challenge arise to find a general solution. The writer utilizes Number Theoretic Transform and Binary Exponentiation to design an efficient algorithm to solve it. While advanced math knowledge, especially Roots of Unity, also took part in the calculation.**

***Keywords*—Number Theoretic Transform, Binary Exponentiation, Periodic Combination Sum, Roots of Unity, Divide and Conquer**

I. Introduction

Mathematical problems are fascinating. What looks as a time wasting simple question to some might actually be a source of knowledge to others. Sometimes, a math problem must involve a clever solution, it might even require a completely unexpected bridge to cross. Such as the following problem we will dive in involves a field of math that seems unrelated, but instead reveals deeper connection of fields that intertwine math as one.

The following math problems is not an original one, in fact the non-general problem is considered a frequent one in math Olympiad. The writer had seen the simpler question on a local math Olympiad back when he was younger. A variation was also found from Canada Mathematical Olympiad 2006, the national mathematical Olympiad for Canadian highschooler in selection for the IMO (International Math Olympiad). Quoting from a handout by Math Olympiad medalist Evan Chen, the non-generalized problem is a classic in math Olympiad world, which goes as the calculating following [1].

$$\sum_{k\geq 0}\binom{1000}{3k}$$

As you can see, the summation involves combination with a constant number of elements, which is 1000. The selection itself is periodic for every term, which is a multiple of 3. This question is quite interesting, and not too hard for a talented highschooler to solve in a math Olympiad. But, this paper will go to a further extent. The writer aims to solve the generalized version of above question, with no restrictions on the combination number of elements nor the modulo of selection. Thus, our new problem looks more or less like what follows.

$$\binom{N}{R}+\binom{N}{M+R}+\binom{N}{2M+R}+\cdots$$

or in a simpler sigma notation

$$\sum_{k\geq 0}\binom{N}{MK+R}$$

Where N is a positive integer signifying the number of objects. The selection is a linear congruence $MK+R$ which means it's congruent to R modulo M. In other words, the selection must be periodic by M. Note that the summation might be very large. Thus, we will find the solution under a certain modulo, the prime 998244353 works well enough.

For a computer science context, the following sum can be computed naively in a quite slow time complexity. However, using some advanced math knowledge we can simplify our problem first. In this paper, the writer will utilize Roots of Unity Filtering to achieve a simpler form of the problem. Then, the writer implements 2 divide and conquer algorithms, which is Number Theoretic Transform and Binary Exponentiation, to speed up the complex calculation.

II. Theoretical Basis

A. Roots of Unity

Roots of unity is a special type of number in the complex domain. Mind that complex number is in the form $a+bi$ where a and b are both real numbers while i is the well known imaginary number. Complex number are usually visualized as point at the 2 dimensional space of real numbers. Where the real line is a horizontal axis while the imaginary is vertical. Complex numbers also be formally written in it's polar form

$$r(\cos x+i\sin x)=re^{ix} \tag{1}$$

Where the right hand side is also known as euler's formula. Also note that *r* is the distance of the point from origin (0,0) while *x* is the magnitude of the point from positive real line.

The n-th roots of unity is defined as all complex z that satisfies $z^n = 1$ [2]. It's easy to notice that r must always be 1. Furthermore by by using trigonometric properties

$$cos(2\pi p) + i\, sin(2\pi p) = 1 = z^n \tag{2}$$

$$e^{\frac{2\pi i p}{n}} = z \tag{3}$$

For all integer n, p<n. Geometrically, the n-th Roots of unity are n equally distanced point in the unit circle where one of the point is in (1,0) since *z=1* is the trivial solution.

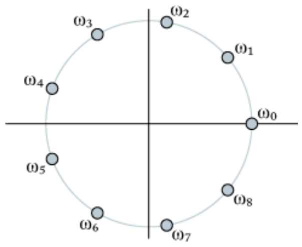


Fig 2.1. Ninth roots of unity

(Taken from kylem.net)

B. *Polynomial*

In this paper, we will use single variable polynomials only. This is an expression involving coefficients that's in the form

$$P(x) = a_n x^n + a_{n-1}x^{n-1} + \cdots + a_2 x^2 + a_1 x + a_0 \tag{4}$$

Where a_i represents real coefficients for $0 \leq i < n$. The above polynomial is said to have a degree *n*, which is the largest power among all terms. A polynomial can be evaluated by using any complex $x = x_0$, which returns $P(x_0)$. Based on Vandermonde Form, we know that any (n+1) points uniquely define a degree n polynomial [3]. Thus, any polynomial can be represented in 2 ways. By a list of coefficients $a_0, a_1, \dots, a_n$ or by list of values it evaluates $P(\mathrm{x}_0), P(\mathrm{x}_1), \dots, P(\mathrm{x}_\mathrm{n})$.

One of the most important operation for polynomials are multiplication. For 2 polynomials P(x) and Q(x), both having degree n, it's multiplication denoted as $(P \cdot Q)(x)$ computed by convoluting both of their coefficients. This operation returns a degree *2n* polynomial. The usual time complexity of this operation is $O(n^2)$, but using algorithms such as Fast Fourier Transform or Number Theoretic Transform we can optimize polynomial multiplication into $O(n\, log\, n)$ time.

C. *Divide and Conquer*

The Divide and Conquer is a problem solving strategy for designing efficient solution. This strategy is proven well to solve many problems in Computer Science, even for designing new optimized algorithms. The strategy relies on 3 phases, which is divide, conquer, and combine. For a small problem, we can usually conquer it directly without the need of recursive steps. This is usually called the base case. On the other hand, larger cases might need to be divided recursively until a small enough subproblem reached.

After a problem divided to it's subproblem, we must solve the subproblems independently. After such, we can finally combine both solution into one that solves the original. The recursion of Divide and Conquer usually described by an algorithmic recurrence *T(n)*. It can be seen as the worst case running time of a Divide and Conquer algorithm of size *n*. From any recurrence *T(n)*, with a driving method *f(n)* of complexity $O(n^c)$, time complexity can be calculated using Master's Theorem as follows

$$T(\mathrm{n}) = aT\left(\frac{\mathrm{n}}{\mathrm{b}}\right) + f(\mathrm{n}) \tag{5}$$

$$T(n) = \begin{cases} \Theta(n^{\log_b a}), & c < \log_b a \\ \Theta(n^{\log_b a} \log n), & c = \log_b a \\ \Theta(f(n)), & c > \log_b a \end{cases} \tag{6}$$

III. Binary Exponentiation & NTT

The following 2 algorithms will be the backbone of the solution for our problem. Thus, the writer decided to touch on them with more detail below. Note that these 2 algorithms share a similarity of being an implementation of Divide & Conquer on it's roots. Let's consider each algorithm and their complexity.

A. *Binary Exponentiation*

Consider the problem of exponentiating a number or polynomial by a huge exponent. Naively multiplying the same thing over and over gives a *O(nf(x))* time complexity for a power *n* and *f(x)* as the complexity of multiplying a same 2 instance. Let's look at an optimization by divide and conquer. The main idea is to split the problem into 2 equal subproblem of smaller exponents. For instance we want to calculate a^b with a and b integers (a can be a polynomial too). Notice that

$$a^b = a^{\frac{b}{2}} \cdot a^{\frac{b}{2}} \tag{1}$$

If *b* is even, the division would be fair. But for *b* odd, the division would not be equal, thus a multiplication with a is needed

$$\mathrm{a}^\mathrm{b} = \mathrm{a}^{\frac{\mathrm{b}-1}{2}} \cdot \mathrm{a}^{\frac{\mathrm{b}-1}{2}} \cdot a \tag{2}$$

It's easy to see that this division would be a recurring step. The base case on itself is quite trivial since a power of 1 is itself. Therefore we have a recurrence function as [4]

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

(3)

Here f(n) denotes the combining process of subproblem solution, for integers its O(1) while for polynomial it's O(n log n) using NTT. If we are dealing with integers, applying master's theorem we have $c = \log_b a = 0$. This concludes the complexity O(log n). On the other hand, for polynomial with degree d case we have $c > 1 = \log_b a$. Thus, we have $T(n) = \Theta(dlogd)$, but since there are $log\, n$ levels of recurrence calls the actual complexity is $\Theta(\text{dlogdlogn})$.

Remember that we are searching solution modulo 998244353, so each step of multiplication must be accompanied with the modulo operation. To further optimize implementation, we may reduce function calls by doubling the power of a at every step while considering the binary representation of b. If the i-th digit of b is on then we calculate the result by a^{2^i}. Eventually we will reach the same solution and complexity.

B. *Number Theoretic Transform*

This variation of Fast Fourier Transform mostly differs at the field they operate. Number Theoretic Transform only works for integer, and calculated under a ring of modulo. Thus, NTT doesn't use the traditional roots of unity. Instead, it acts under n-th roots of unity in the modular arithmetic field Z_p where $n \mid p-1$. NTT provides a more accurate calculation to FFT with modulo since it already operates on integer, but a bit slower due to many modulo operation needed.

Let's say we want to calculate P(x) with Q(x), both with degree n. The heart of this algorithm is evaluating n+1 values of complex numbers to the polynomial, which is $P(\mathrm{x}_0), P(\mathrm{x}_1), \ldots, P(\mathrm{x}_\mathrm{n})$, in only O(n log n) time, this is NTT. After we have $Q(\mathrm{x}_0), Q(\mathrm{x}_1), \ldots, Q(\mathrm{x}_\mathrm{n})$, we can calculate each corresponding integer value to get $P\mathrm{Q}(\mathrm{x}_0), P\mathrm{Q}(\mathrm{x}_1), \ldots, P\mathrm{Q}(\mathrm{x}_\mathrm{n})$, in only O(n) time. After which we also figure out the values $P\mathrm{Q}(\text{-x}_0), P\mathrm{Q}(\text{-x}_1), \ldots, P\mathrm{Q}(\text{-x}_\mathrm{n})$. Now that we have 2n points, we can interpolate these points to get the multiplied polynomial with degree 2n, the interpolation done with inverseNTT, a slight variation to NTT, and it's also O(n log n) [5].

So how does this NTT algorithm actually works, and what does roots of unity do with it. To do NTT to P(x). The points that we use are the n-th root of unity, which is $\omega, \omega^2, \omega^3, \ldots$. We must first divide P(x) into 2 polynomials with n/2 degrees, they are the odd and even coefficient polynoms.

$$P_{even}(x) = a_0 + a_2x^2 + a_4x^4 + \cdots$$

$$P_{odd}(x) = a_1x + a_3x^3 + a_5x^5 + \cdots$$

Where $P(x) = P_{even}(x^2) + xP_{odd}(x^2)$. What we need right now is how to combine the values of NTT from $P_{even}(x)$ and $P_{\mathrm{odd}}(x)$ to achieve NTT of P(x).

The first $\frac{n}{2}$ values are just an evaluation to the form above.

$$P(\omega^k) = P_{even}(\omega^{2\mathrm{k}}) + \omega^k P_{odd}(\omega^{2\mathrm{k}})$$

And the next $\frac{n}{2}$ values are surprisingly simple due to roots of unity.

$$P(\omega^{k+n/2}) = P_{even}(\omega^{2\mathrm{k}}) - \omega^k P_{odd}(\omega^{2\mathrm{k}})$$

Due to the similarity, the NTT can be easily determined. The recurrence is denoted by

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

By master's theorem, this is O(n log n) time. The inverse of NTT is surprisingly simple too. The inverse is just the inverse of NTT Vandermonde matrix [3]. It can be proven that it's in the form $\frac{\omega^{-kj}}{n}$ instead of NTT Vandermonde matrix's ω^{kj}. Therefore, the inverse NTT is just the same as applying NTT again with a slight variation, and it's complexity is similar. So, we can conclude that NTT in total have a complexity O(n log n) time.

IV. Mathematical Solution

Calculating the sum of periodic binomials is quite heavy to do naively. To ease things up, we will utilize roots of unity to find a simplification to our problem. Restating what was said before, the main calculation is

$$\sum_{k\geq 0} \binom{N}{MK+R} \; mod\; 998244353$$

The choice of modulus is a large prime, reasons of why would be clear later. Let's consider the first one of the M-th roots of unity, that is a single complex number ω such $\omega = e^{\frac{2\pi i}{M}}$. But before we actually solve the problem there is a lemma regarding roots of unity that we need to address.

A. *Roots of unity sum lemma*

It will be proven that if an integer x is not divisible by M the following must hold

$$1 + \omega^x + \omega^{2x} + \cdots + \omega^{(M-1)x} = 0$$

(1)

Start from the obvious fact that is $\omega^{Mx} = 1$. Factor out

$$(\omega^x - 1)(\omega^{x(M-1)} + \omega^{x(M-2)} + \cdots + \omega^x + 1) = 0$$

(2)

Notice either one of the 2 polynomial must be 0. But, if M doesn't divide x there is no way

$$\omega^x = 1 \rightarrow \omega^x - 1 = 0$$

(3)

Thus, in this case the equality must hold and lemma is proven.

B. *Generating Function*

A certain generating function would be used to solve this problem. Let $f(x) = (x+1)^n$ for any complex x. By expanding the binomial coefficient we have

$$f(x) = \binom{N}{0} + \binom{N}{1}x + \binom{N}{2}x^2 + \cdots + \binom{N}{N}x^N$$

(1)

It's trivial that for *x=1* we have

$$2^n = f(1) = \binom{N}{0} + \binom{N}{1} + \cdots + \binom{N}{N} \tag{2}$$

We also define another function S(R) where

$$S(R) = \sum_{k=0} \binom{N}{Mk+R} \tag{3}$$

C. *Roots of unity Filter*

What we want to calculate is actually S(R). First, let's search for a simpler form of S(0). Notice and clarify the following

$$f(\omega^k) = (\omega^k + 1)^n = \sum_{i<N} \binom{N}{i} \omega^{ki}$$

$$f(\omega^k) = \sum_{i=0}^{M-1} S(i)\omega^{ki} \tag{4}$$

The key is to sum over all from *k* equals 0 to *M-1*

$$\sum_{j=0}^{M-1} f(\omega^j) = \sum_{j=0}^{M-1}\sum_{i=0}^{M-1} S(i)\omega^{ji} \tag{5}$$

Switch the order of sigma summation into

$$\sum_{j=0}^{M-1} f(\omega^j) = \sum_{i=0}^{M-1}\sum_{j=0}^{M-1} S(i)\omega^{ji} \tag{6}$$

This is where we apply our lemma. Since every sum of roots of unity where the power is not divisible by M leads to 0, all terms with *i* not divisible by M are filtered. Thus, only terms with *i=0* persists. Therefore we are left with

$$\sum_{j=0}^{M-1} f(\omega^j) = \sum_{j=0}^{M-1} S(0) = M \cdot S(0) \tag{7}$$

And this concludes our search for a simpler equation of S(0).

D. *Simplified Form*

We just found out that

$$S(0) = \frac{2^N + (\omega+1)^N + (\omega^2+1)^N + \cdots + (\omega^{M-1}+1)^N}{M} \tag{8}$$

Now what about S(R)? We need to make sure that during the summation of roots of unity, only the term *i=R* persists. This can be done by multiplying $\omega^{(M-R)k}$ for every $f(\omega^k)$. Thus, for a general R our simplified S(R) is

$$\frac{2^N + (\omega+1)^N\omega^{M-R} + \cdots + (\omega^{M-1}+1)^N\omega^{(M-R)(M-1)}}{M} \tag{9}$$

V. Implementation

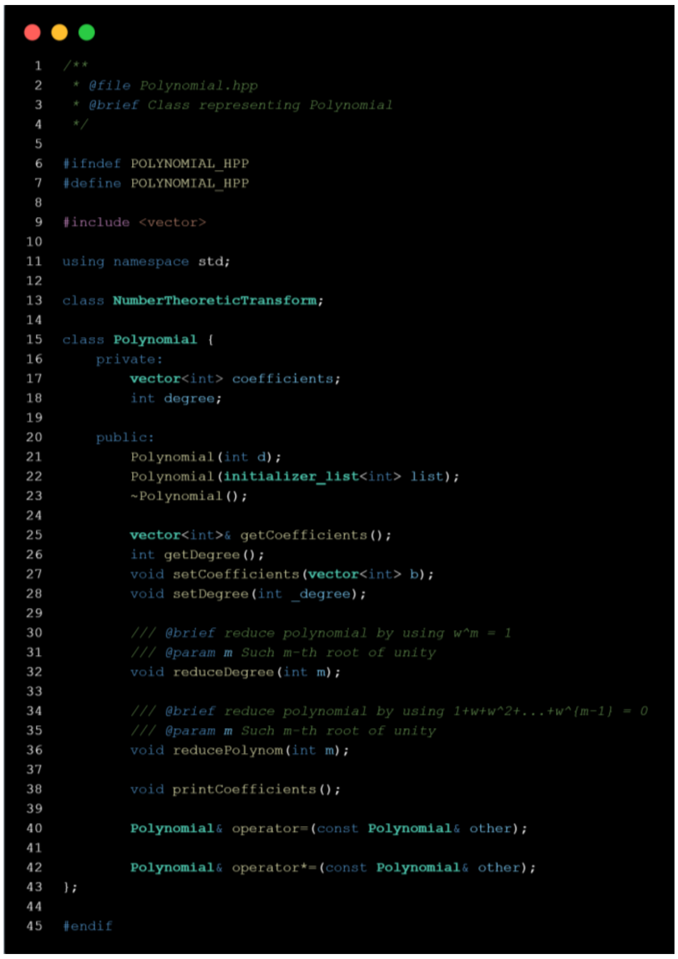
Since the problem is simplified already, we are ready to implement it as a code. Firstly, note that we are not actually using the roots of unity as a complex number. The main reason is due to the amount of modulo operation we must do, the use of complex numbers might lead to unwanted inaccuracy. So instead, we are using n-th root of unity under primitive field p where a ω satisfies $\omega^n \equiv 1 \bmod p$ but $\omega^k \not\equiv 1 \bmod p$ for all *k<n*. Since we use modulus 998244353 we pick ω=3 since the order of 3 modulo 998244353 is exactly 998244352 as we wanted.

To handle the actual properties of ω^M applies under modulo 998244353, we must make sure M is less than that prime so the roots of unity always exist. Notice that the property of the lemma is no longer true for this root, although it is crucial. Thus, we will do it manually by utilizing

$$1 + \omega + \omega^2 + \cdots + \omega^{(M-1)} = 0$$

Every coefficient on the polynomial of interest must be reduced by the minimum coefficient. Same as for the property of $\omega^M = 1$ we can handle it by reducing every polynomial of interests into a degree *M-1* polynomial with the coefficient accumulated. This way, we won't ever actually have a polynomial larger than degree *M-1.* This will speed up our NTT and Polynomial Exponentiation significantly.

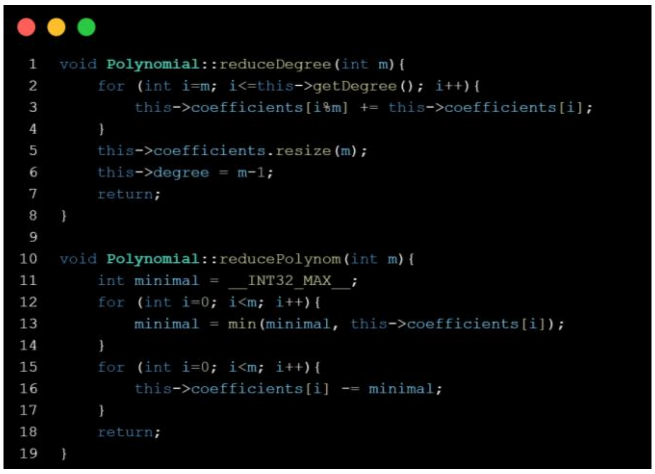
The Writer implemented this algorithm using C++. The choice of language is for speed optimalization since this language is known for effectiveness. Further details about the implementation can be seen on the following Github repository. The polynomial is implemented as an object which defined as



```cpp
/**
 * @file Polynomial.hpp
 * @brief Class representing Polynomial
 */

#ifndef POLYNOMIAL_HPP
#define POLYNOMIAL_HPP

#include <vector>

using namespace std;

class NumberTheoreticTransform;

class Polynomial {
    private:
        vector<int> coefficients;
        int degree;

    public:
        Polynomial(int d);
        Polynomial(initializer_list<int> list);
        ~Polynomial();

        vector<int>& getCoefficients();
        int getDegree();
        void setCoefficients(vector<int> b);
        void setDegree(int _degree);

        /// @brief reduce polynomial by using w^m = 1
        /// @param m Such m-th root of unity
        void reduceDegree(int m);

        /// @brief reduce polynomial by using 1+w+w^2+...+w^{m-1} = 0
        /// @param m Such m-th root of unity
        void reducePolynom(int m);

        void printCoefficients();

        Polynomial& operator=(const Polynomial& other);

        Polynomial& operator*=(const Polynomial& other);
};

#endif
```

Fig 5.1. Polynomial Object Definition

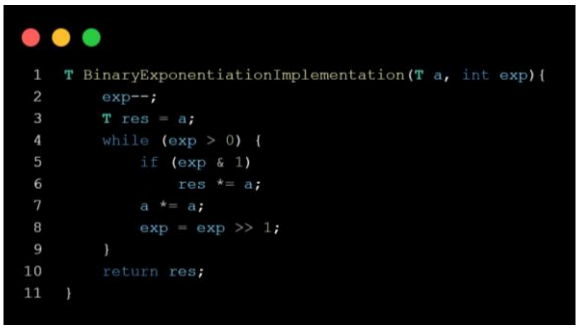
As explained before, since we are using integer roots over a primitive field we must manipulate the properties of ω such it behave like the roots of M. This is done manually by 2 procedures. reduceDegree() will accumulate all coefficients with the same power modulo M such the resulting polynomial degree never exceeds M-1. While the procedure reducePolynom() will utilize the lemma to make sure the coefficients are minimized. Their implementation is provided in the snippet below



```cpp
void Polynomial::reduceDegree(int m){
    for (int i=m; i<=this->getDegree(); i++){
        this->coefficients[i%m] += this->coefficients[i];
    }
    this->coefficients.resize(m);
    this->degree = m-1;
    return;
}

void Polynomial::reducePolynom(int m){
    int minimal = __INT32_MAX__;
    for (int i=0; i<m; i++){
        minimal = min(minimal, this->coefficients[i]);
    }
    for (int i=0; i<m; i++){
        this->coefficients[i] -= minimal;
    }
    return;
}
```

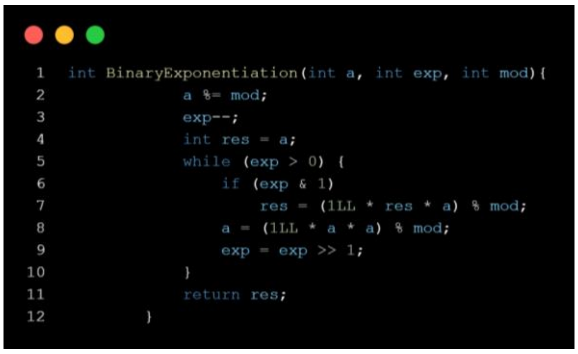
Fig 5.2. Snippet of Polynomial procedures

The algorithms are not that hard to implement either. Here is the code snippet for Binary Exponentiation



```cpp
T BinaryExponentiationImplementation(T a, int exp){
    exp--;
    T res = a;
    while (exp > 0) {
        if (exp & 1)
            res *= a;
        a *= a;
        exp = exp >> 1;
    }
    return res;
}
```

Fig 5.3. Polynomial Exponentiation Implementation



```cpp
int BinaryExponentiation(int a, int exp, int mod){
            a %= mod;
            exp--;
            int res = a;
            while (exp > 0) {
                if (exp & 1)
                    res = (1LL * res * a) % mod;
                a = (1LL * a * a) % mod;
                exp = exp >> 1;
            }
            return res;
        }
```

Fig 5.4. Integer Exponentiation Implementation

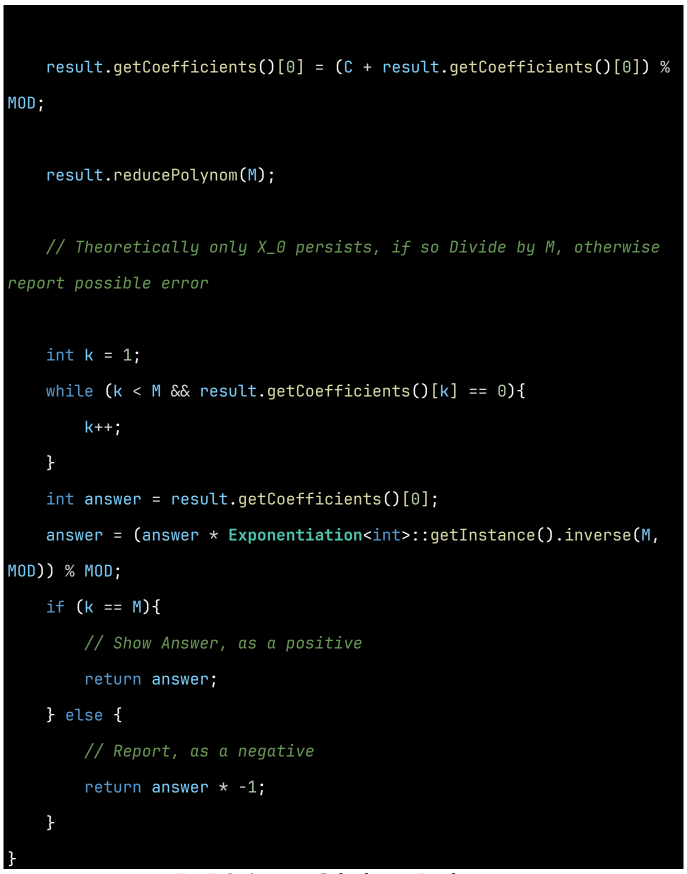
See that the algorithms is almost exact for both polynomial and integers. As it was said before, this code optimizes the explained implementation by using iterative approach instead of recurrence one. This is done by exploiting the binary of the exponent to decide what should be multiplied next. The multiplication for Polynomial uses Number Theoretic Transform which is implemented as the following

```
void NumberTheoreticTransform::ntt(vector<int>& a, int n, int root) {
    if (n == 1) return;
    vector<int> even(n / 2), odd(n / 2);
    for (int i = 0; i < n / 2; ++i) {
        even[i] = a[2 * i];
        odd[i] = a[2 * i + 1];
    }

    NumberTheoreticTransform::ntt(even, n / 2, root);
    NumberTheoreticTransform::ntt(odd, n / 2, root);

    int wlen =
Exponentiation<int>::getInstance().BinaryExponentiation(root, (MOD -
1) / n, MOD);
    int w = 1;

    for (int i = 0; i < n / 2; ++i) {
        a[i] = (even[i] + 1LL * w * odd[i] % MOD) % MOD;
        a[i + n / 2] = (even[i] - 1LL * w * odd[i] % MOD + MOD) % MOD;
        w = 1LL * w * wlen % MOD;
    }
    return;
}

void NumberTheoreticTransform::ntt_inverse(vector<int>& a, int n) {
    ntt(a, n,
Exponentiation<int>::getInstance().BinaryExponentiation(ROOT, MOD - 2,
MOD));
    int inv_n =
Exponentiation<int>::getInstance().BinaryExponentiation(n, MOD - 2,
MOD);
    for (int i = 0; i < n; ++i) {
        a[i] = 1LL * a[i] * inv_n % MOD;
    }
    return;
}

Polynomial NumberTheoreticTransform::multiply(Polynomial a, Polynomial
b) {
    int n = 1;
       while (n < a.getDegree() + b.getDegree() + 1) {
        n <<= 1;
    }
    a.getCoefficients().resize(n);
    b.getCoefficients().resize(n);
    NumberTheoreticTransform::ntt(a.getCoefficients(), n, ROOT);
    NumberTheoreticTransform::ntt(b.getCoefficients(), n, ROOT);
       vector<int> c(n);
    for (int i = 0; i < n; ++i) {
        c[i] = 1LL * a.getCoefficients()[i] * b.getCoefficients()[i] %
MOD;
    }
    NumberTheoreticTransform::ntt_inverse(c, n);
    Polynomial res(n-2);
    res.setCoefficients(c);
    res.reduceDegree(m);
    res.reducePolynom(m);
    return res;
}
```

Fig 5.5. Number Theoretic Transform Implementation

As you can see the procedure reduceDegree() and reducePolynom() are called after each NTT multiplication. This way every resulting polynomial is minimized degree wise and the next NTT will be significantly faster. Finally to piece everything together, there is a Solver object that utilizes all the algorithms above to calculate the answer like the simplified form we discovered before.

```
int Solver::Calculate(int N, int M, int R){
    if (R==0){
        R = M;
    }
    NumberTheoreticTransform::getInstance().setM(M);
    int C = Exponentiation<int>::getInstance().BinaryExponentiation(2,
N, MOD);
    Polynomial result(M-1);
    Polynomial temp(0);
    Polynomial expo(M-1);
    temp.getCoefficients()[0] = 1;
    temp.getCoefficients().push_back(1);
    expo =
Exponentiation<Polynomial>::getInstance().BinaryExponentiation(temp,
N);
    for (int j=0; j<M; j++){
        result.getCoefficients()[(j + M - R) % M] +=
expo.getCoefficients()[j];
    }
    for (int i=2; i<M; i++){
        temp.getCoefficients().push_back(1);
        temp.getCoefficients()[i-1] = 0;
        expo =
Exponentiation<Polynomial>::getInstance().BinaryExponentiation(temp,
N);
        for (int j=0; j<M; j++){
                    result.getCoefficients()[(j + (i * (M-R))) % M] +=
expo.getCoefficients()[j];
        }
    }
```



```
    result.getCoefficients()[0] = (C + result.getCoefficients()[0]) %
MOD;

    result.reducePolynom(M);

    // Theoretically only X_0 persists, if so Divide by M, otherwise
report possible error

    int k = 1;
    while (k < M && result.getCoefficients()[k] == 0){
        k++;
    }
    int answer = result.getCoefficients()[0];
    answer = (answer * Exponentiation<int>::getInstance().inverse(M,
MOD)) % MOD;
    if (k == M){
        // Show Answer, as a positive
        return answer;
    } else {
        // Report, as a negative
        return answer * -1;
    }
}
```

Fig 5.6. Answer Calculation Implementation

To calculate S(R), the multiplication with $\omega^{(M-R)k}$ is handled with indexing [(j + (i * (M-R))) % M] which exploits the cyclic nature of powers with $\omega^M = 1$. Pay attention to the final reducePolynom() call. At that point we are at the final calculated polynomial. Theoretically, every coefficient of ω at that polynomial will be equal. Thus, we can reduce it one last time and ended up with a single integer, which divided by M is our answer. But in reality, sometimes the coefficient are not all equal, which gives doubt to the final answer since a root of unity persists in the answer while it's a complex number. This is why the writer notifies the user of a probably miscalculation if such thing happens.

Finally we must make sure that the division by M is correct under modulo. By Fermat's little Theorem, we have

$$M^{-1} \equiv M^{p-2} \bmod p$$

This is why choosing 998244353 as our modulus gave an advantage. Since selecting composite modulus complicates us by needing Chinese Remainder Theorem to calculate inverse. In the above code, the inverse is just

$$3^{998244351} \bmod 998244353$$

Which can be calculated in O(log n) time using Binary Exponentiation.

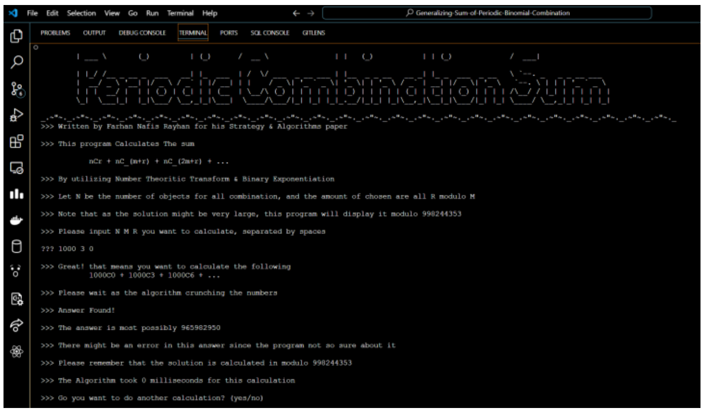
VI. Analysis

The program runs pretty quick with ability to calculate up to N = 100000. Above that the programs seems to run out of memory and thus cannot continue the calculation. For N=10000, by average the solution found in about a minute. Based on the algorithms time complexity, the total complexity of the program should be $O(m^2 logm\ logn)$. Which isn't quite reflected by the reality. Truth is, the program could be way slower do to the heavy computations of the modulo.

The main problem of the program currently is the low of accuracy. The case where root coefficient not equal is highly probably. The program also works very well for small N but once it reaches the thousands it tends to gave a wrong answer even when no indication reported. One such case is for

$$\sum_{k\geq 0} \binom{1000}{3k}$$

Where in the reference mentioned before the solution is $\frac{2^{1000}-1}{3}$ which should've been 7742092 in modulo 998244353. Yet the program returns 965982950


File Edit Selection View Go Run Terminal Help
Generalizing-Sum-of-Periodic-Binomial-Combination
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS SQL CONSOLE GITLENS
Periodic Combination Sum

```
>>> Written by Farhan Nafis Rayhan for his Strategy & Algorithms paper

>>> This program Calculates The sum

        nCr + nC_(m+r) + nC_(2m+r) + ...

>>> By utilizing Number Theoritic Transform & Binary Exponentiation

>>> Let N be the number of objects for all combination, and the amount of chosen are all R modulo M

>>> Note that as the solution might be very large, this program will display it modulo 998244353

>>> Please input N M R you want to calculate, separated by spaces

??? 1000 3 0

>>> Great! that means you want to calculate the following
        1000C0 + 1000C3 + 1000C6 + ...

>>> Please wait as the algorithm crunching the numbers

>>> Answer Found!

>>> The answer is most possibly 965982950

>>> There might be an error in this answer since the program not so sure about it

>>> Please remember that the solution is calculated in modulo 998244353

>>> The Algorithm took 0 milliseconds for this calculation

>>> Go you want to do another calculation? (yes/no)
```

Fig 6.1. Classical Test Case

This is far from the solution and instead was closer to the modulo 998244353 instead. There's a high probability that an integer overflow happened and not handled yet, which explains most of the error.

VII. Conclusion

In conclusion, the program written by the author is not perfect. There is still a huge room for improvement on the implementation side. But, it still proves that the problem can be generalized using Number Theoretic Transform and Binary Exponentiation. Plus, utilizing roots of unity a simpler form can be achieved. Anyway, the author is still grateful of the knowledge achieved while working on this paper.

Acknowledgment

The writer would like to thank first of all, to God for giving me all ability and chance to finish this paper. I also would like to express our sincere gratitude to Dr. Ir. Rinaldi Munir, M.T. for his guidance throughout the development of this paper and the teachings as my Strategy & Algorithms lecturer throughout this semester. I am also grateful to Institut Teknologi Bandung for providing the computational resources necessary and giving the writer a chance to finish his second paper. Appreciation also extends to the community of Math Olympiad participants and enthusiast since they are the reason why the problem discussed here are known. Lastly, I thank my family, friends, and everyone providing support while writing this paper

References

[1] E. Chen, "Summation", evanchen.cc, October 13, 2023. [Online]. Available: https://web.evanchen.cc/handouts/Summation/Summation.pdf

[2] R. Li, "Roots of Unity," Contest Lectures, Department of Computer Science, Stanford University, Stanford, CA, [Online]. Available: https://cs.stanford.edu/~rayyli/static/contest/lectures/Ray%20Li%20rootsofunity.pdf

[3] A. Kotelnikov, "The Vandermonde Determinant: A Novel Proof," Towards Data Science, March 24, 2021. [Online]. Available: https://towardsdatascience.com/the-vandermonde-determinant-a-novel-proof-851d107bd728

[4] J. Kogler, "Binary Exponentiation", cp-algorithms, September 12, 2023. [Online]. Available: https://cp-algorithms.com/algebra/binary-exp.html

[5] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, Introduction to Algorithms, 4th ed. Cambridge, MA: MIT Press, 2022.

PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 12 Juni 2024

[illegible]

Farhan Nafis Rayhan - 13522037