AN EFFICIENT APPROACH TO CALCULATE FACTORIALS IN PRIME FIELD

Rully Soelaiman[1,*], Ferdinand Jason Gondowijoyo[2], M.M. Irfan Subakti[3], and Yudhi Purwananto[4]

[1, 2, 3, 4]Institut Teknologi Sepuluh Nopember, Surabaya, Indonesia
*Corresponding Author

ABSTRACT
Factorial is extremely important in a combinatorial calculation. E.g., there are $n!$ ways to arrange n numbers in different orders. One of the uses of factorial is to find the binomial coefficient, i.e., the number of ways of picking k unordered outcomes from n possibilities. Aside from combinatorial calculation, factorial also has a role in calculus (Taylor's theorem), probability theory, and network security that use the prime field domain. But, another problem arose, i.e., how to efficiently compute the factorial. This paper will describe a new approach for efficiently calculating factorials in the prime field. Two testing models are involved in this paper: the correctness test by source-code submission to Sphere Online Judge, and the performance test by generating their chart of runtimes. Based on the testing result from the given case studies, it turns out that our approach has a better performance compared to the Multipoint Evaluation algorithm adaptation approach.

Keywords: *factorial, shifting evaluation values, number-theoretic transform, number theory.*

1. INTRODUCTION
In mathematics, the factorial of a positive integer *n*, denoted by *n!* is the product of all positive integers less than or equal to *n*, as in (1), except for *n = 0*, where *0! = 1*.

$$n! = \prod_{i=1}^{n} i \tag{1}$$

There are several motivations for this definition, and one of them is there is exactly one permutation of zero objects (with nothing to permute, the only rearrangement is to do nothing). Factorials were used to count permutations at least as early as the 12th century, by Indian scholars. In 1677, Fabian Stedman described factorials as applied to change ringing, a musical art involving the ringing of many tuned bells. The notation *n!* was introduced by the French mathematician Christian Kramp in 1808 [1]. The factorial operation is encountered in many areas of mathematics, notably in combinatorics, algebra, and mathematical analysis. Its most basic use is a permutation, which counts the possible distinct sequences of n distinct objects: there are *n!*. Although the factorial function has its roots in combinatorics, formulas involving factorials occur in many areas of mathematics. For example, in encoding a number, a factorial number system or *factoradic* used factorial as the base of calculation. *factoradic* is a method of encoding a number by expressing a number as the sum of multiples of factorials with the rule that you cannot use a coefficient larger than *n* for *n!* term. For example, *2020* when expressed as a *factoradic* is $2 \times 6! + 4 \times 5! + 4 \times 4! + 0 \times 3! + 2 \times 2! + 0 \times 1! + 0 \times 0!$ or (2 4 4 0 2 0 0). If efficiency is not a concern, computing factorials are trivial from an algorithmic point of view: successively multiplying a variable initialized to *1* by the integers up to *n* will compute $n!$ provided the result fits in the variable. The main practical difficulty in computing factorials is the size of the result. To assure that the exact result will fit all legal values of even the smallest commonly used integral type (8-bit signed integers) would require more than 700 bits. The values 12! and 20! are the largest factorials that can be stored in, respectively, the 32-bit and 64-bit integers commonly used in personal computers. Most software applications will compute small factorials by direct multiplication or a table lookup. Larger factorial values can be approximated using Stirling's formula, but in many cases, we need the exact value of the factorials. In mathematics, a finite field is a field that contains a finite number of elements. As with any field, a finite field is a set on which the operations of multiplication, addition, subtraction and division are defined and satisfy certain basic rules. The most common examples of finite fields are given by the integers mod P when P is a prime number. The storage problem would be solved using finite field, but another problem will arise, calculate a large factorial, also takes a massive amount of time because the complexity is $\mathcal{O}(\mathrm{n})$. For example, if we want to calculate the value of $10^{11}!$ in some finite fields with the usual method, it takes about 16 minutes to complete. Until now, there has been no method or algorithm for calculating factorials with complexity below $\mathcal{O}(\mathrm{n})$. However, several methods are known to accelerate the calculation of factorial using Wilson's theorem [2] or optimization in multiplication using GMP [3]. Yet, J. Gauthier [4] proposed a fast method to evaluate the polynomial at *n* arbitrary point, which can be adjusted to calculate factorial, named Multipoint Evaluation. The adaptation of this algorithm will run with complexity $\mathcal{O}\left(\sqrt{\mathrm{n}}\log^2\sqrt{\mathrm{n}}\right)$ (around 25 seconds for 10^{11}!), fast enough, but it has some issues as follows:

1. This method has dependencies on polynomial operations, and the heaviest part is polynomial division,
2. This method requires us to store every constant of the degree polynomial across the evaluation of tree nodes, which results in the large memory,
3. This method will run longer than Horner's method if the size or less than 256,
4. The algorithm assumed that will be given $\mathrm{n} = 2^{\mathrm{k}}$ unique evaluation points. Suppose n is not a power of two. In general, we have two choices. We may either add phantom points to round up to the next power of two, or we may adjust the code to have an almost binary tree [4].

According to [5], there are many methods to optimize the calculation of polynomial division, but the calculation still has high complexity. The main idea of our approach is rather than evaluating the polynomial, we interpolating the polynomial using the Shifting Evaluation Values algorithm recursively. This method is only dependent on the multiplication operation on the polynomial. With the experiments and testing that has been done, our approach was able to run with complexity $\mathcal{O}\left(\sqrt{\mathrm{n}}\log\sqrt{\mathrm{n}}\right)$ (around 3 seconds for 10^{11}!). This paper is organized as follows. In Section 2, the problem is introduced. Section 3 explains the algorithms used in our approach. Section 4 talks about our approach to solving the problem. In section 5, the evaluation method and results are described, here we will compare our approach with the Multipoint Evaluation algorithm adaptation. Conclusions are presented in section 6.

2. PROBLEM STATEMENT

In this paper, we present a design solution to the factorial computation problem in the finite field on a case study of Sphere Online Judge (SPOJ) Factorial Modulo Prime (FACTMODP) [6]. SPOJ is an online judge, problem set, and contest hosting service. This online judge accepts over 40 programming languages. SPOJ consists of five problem categories, there is tutorial, classical, challenge, partial, and riddle. FACTMODP is a problem from the classical section. Our solution will be checked with some test cases. If we can pass all the test cases, then an accepted verdict will come out. There are many verdicts that we can receive, namely, the wrong answer, run time error, time limit exceeded, compilation error, etc. At this problem, we have to find the value of

$$N!\ \bmod P \tag{2}$$

The problem has constraints as follows:

1. $N(0 \leq N \leq 10^{11})$
2. $P(2 \leq P \leq 10^{11}, \in \text{prime})$
3. The runtime limit program execution time limit is 10 - 15 seconds per test file
4. One test file will contain about $\mathrm{T}(1 \leq 10^5)$ test case
5. Is guaranteed that for every file $\sum \sqrt{P} \leq 3.2 \times 10^5$.

For comparison, using the normal method and considering the capability of the SPOJ server is 10^8 instructions per second, to calculate the value of the equation (2) with $\mathrm{N} = 10^{10}$ and $\mathrm{P} = 99999999977$ takes about 100 seconds, which will exceed the time limit given.

```
Algorithm 1 Normal Factorial Calculation
Input: $N, P$
Output: $N! \bmod P$
1: function FACTORIAL($N, P$)
2:     $res \leftarrow 1$
3:     for $i \leftarrow 1$ to $N$ do
4:         $res \leftarrow (res \times i) \bmod P$
5:     end for
6:     return $res$
7: end function
```

Figure 1. Algorithm for Normal Factorial Calculation

3. LITERATURE REVIEW

To overcome the above-mentioned problems, several terms and definitions will be defined in the next subsection.

3.1. Shifting Evaluation Values

Let a and $\mathrm{r}_0, \mathrm{r}_1, \cdots \mathrm{r}_\mathrm{d}$ be in R, P is a polynomial and r_i is an arithmetic progression. Given $\mathrm{P}(\mathrm{r}_0), \mathrm{P}(\mathrm{r}_1), \cdots, \mathrm{P}(\mathrm{r}_\mathrm{d})$ and integer a, we can compute $\mathrm{P}(\mathrm{r}_0 + \mathrm{a}), \mathrm{P}(\mathrm{r}_1 + \mathrm{a}), \cdots, \mathrm{P}(\mathrm{r}_\mathrm{d} + \mathrm{a})$ in $\mathcal{O}(\mathrm{M}(\mathrm{d}) + \mathrm{d})$ with $\mathrm{M}(\mathrm{d})$ is the complexity of polynomial multiplication. The $\mathcal{O}(\mathrm{d})$ additional operations come from additional pre-processing and post-processing operations [4]. Before shifting the polynomial, there is a calculation prerequisite that must be met first. First, we need to define $\delta(\mathrm{i}, \mathrm{d})$ and $\Delta(\mathrm{a}, \mathrm{k}, \mathrm{d})$ in equations (3) and (4).

$$\delta(0, d) = \prod_{j=1}^{d} (-j)$$
$$\delta(i, d) = \frac{i}{i - d - 1} \delta(i - 1, d) \tag{3}$$

$$\Delta(a, 0, d) = \prod_{j=1}^{d} (a - j)$$
$$\Delta(a, k, d) = \frac{a + k}{a + k - d - 1} \Delta(a, k - 1, d)\ k \ \&= 1, 2, \cdots, d \tag{4}$$

In the shifting step, defined polynomial $\widetilde{\mathrm{P}}$, S, and Q as in (5) and notation $\mathrm{C}(\mathrm{Q}, \mathrm{k})$ to denote the coefficient of degree k of polynomial Q.

$$\widetilde{\mathrm{P}} = \sum_{\mathrm{i}=0}^{\mathrm{d}} \frac{\mathrm{P}(\mathrm{i})}{\delta(\mathrm{i}, \mathrm{d})} \mathrm{X}^\mathrm{i}$$
$$S = \sum_{i=0}^{2d} \frac{1}{a + i - d} X^i \tag{5}$$
$$Q = P \cdot S$$

The shifted value of polynomial P is shown by (6). It can be noted that polynomial P is not part of the input of this algorithm. All operations made below on integer values take place in R.

$$P(r_i + a) = \Delta(a, i, d) \cdot C(Q, d + i), i = 0 \cdots d \tag{6}$$

3.2. Middle Product

In [8], an operation called the middle product is defined. This is an algorithm for the inverse, division, and square root of power series. The key trick of this algorithm is computing the n middle coefficient of a $(2\mathrm{n} - 1) \times \mathrm{n}$ full product in the same number of multiplications as a full $\mathrm{n} \times \mathrm{n}$ product. Given a polynomial A, B in R with respective degrees d and $2d$. Defined polynomial C showed by (7).

$$PC = A \cdot B = C_0 + C_1 X^{d+1} + C_2 X^{d+2} \tag{7}$$

C_1 is what we call the middle product of polynomial A and B. This is precisely what is needed in (6). Based on [8] in Corollary 3, this operation can be computed in two Fourier transform of size $2n$. With this, we can rewrite the equation (5) and (6) to (8).

$$Q' = MP(P, S)$$
$$P(r_i + a) = \Delta(a, i, d) \cdot C(Q', i) \tag{8}$$

3.3. Chinese Remainder Theorem

Chinese Remainder Theorem is a theorem that gives a unique solution to simultaneous linear congruences with coprime modulo. For our approach to applying to all prime modulo, the Chinese Remainder Theorem can be used to overcome the limitations of our approach. To simplify, we will use a congruence system with $\mathrm{k} = 2$. There is a special case in solving congruence system with $\mathrm{k} = 2$, and n_1 and n_2 coprime each other like in (9).

$$x \equiv a_1 \bmod n_1 \tag{9}$$

$$x \equiv a_2 \bmod n_2$$

By using Bezeout's Identity, which says that there are two number m_1 and m_2 such that (10) is true, and the congruence system formula will turn into (11).

$$m_1 n_1 + m_2 n_2 = 1 \tag{10}$$

$$\begin{aligned} x &\equiv a_1 m_2 n_2 + a_2 m_1 n_1 \\ &\equiv \mathrm{a}_1(1 - \mathrm{m}_1\mathrm{n}_1) + \mathrm{a}_2\mathrm{m}_1\mathrm{n}_1 \\ &\equiv a_1 + (a_2 - a_1) m_1 n_1 \end{aligned} \tag{11}$$

With two base prime numbers, we can calculate factorial with any prime modulo. In the next section, we will explain the limitation when choosing base prime numbers for our approach, and why we need this Chinese Remainder Theorem.

3.4. Number Theoretic Transform

Number Theoretic Transform (NTT) is a discrete Fourier Transform defined over finite fields and rings. NTT is computed in integer domain and hence they are found to be superior compared to already existing FFT [9]. The most important part of the Fourier Transform is

$$\begin{aligned} \omega &= e^{\frac{2\pi i}{n}} \\ \omega^n &= 1 \end{aligned} \tag{12}$$

But, instead of using a complex number ω, we want to do everything in some other number field where $\omega^n = 1$. From Fermat's theorem in [10], we know that

$$a^{P-1} \equiv 1 \pmod P \tag{13}$$

is true and there exists a primitive root g, and integers x in range 1 to $P - 1$, so that g^x mod P goes through all the numbers from 1 to $P - 1$ in some order. This will work if prime number have form $\mathrm{P} = \mathrm{k} \cdot \mathrm{n} + 1$, so that

$$\omega^n \equiv g^{kn} \equiv g^{P-1} \equiv 1 \pmod P \tag{14}$$

With this, Number Theoretic Transform can be implemented using Fast Fourier Transform with generator as ω. Finding a generator is not a simple problem either, as shown in Algorithm 2, the running time of this finding is $\mathcal{O}(\log^6 \mathrm{p})$ (assuming the Generalized Riemann Hypothesis [11]). With the problem statement defined in section 2. there will be additional complexity to find a generator, which is not cheap. To overcome that problem and limitation of our NTT implementation, we use static prime and generator and use the Chinese Remainder Theorem to find the real value under the real modulo. Based on research that has been done, Table 1 shows base prime numbers that work well will modulo under 10^{11}. But further research is needed to determine the limits of these prime numbers.

Table 1. Prime number and the generator under 10^{11}

Prime Number	Generator
709.143.768.229.478.401	31
711.416.664.922.521.601	19

3.5. Montgomery Multiplication

In 1985, American mathematician Peter L. Montgomery proposed a representation of residue classes to speed modular multiplication without affecting the modular addition and subtraction algorithms [12]. Given two integers a and b and modulus N, the classical modular multiplication algorithm computes the double-width product ab, and then performs a division, subtracting multiples of N to cancel out the unwanted high bits until the remainder is once again less than N. This algorithm instead adds multiples of N to cancel out the low bits until the result is a multiple of a convenient

```
Algorithm 2 Primitive Root
Input: $P$
Output: Primitive Root of $P$
function PrimitiveRoot($P$)
    $x \leftarrow \phi(P)$
    $f \leftarrow factor\ of\ x$
    for $g \leftarrow 1$ to $P$ do
        $ok \leftarrow true$
        for $i \leftarrow 1$ to $|f|$ do
            $ok \leftarrow ok\ AND\ (g^{\frac{x}{f[i]}} \mod P! = 1)$
        end for
        if $ok$ then
            return $g$
        end if
    end for
    return $-1$
end function
```

Figure 2. Algorithm for Primitive Root

constant $R > N$, R is coprime to N (possibly the machine word size or power thereof). Then the low bits are discarded, producing a result of less than $2N$. The result is the desired product divided by R, which is less inconvenient than it might appear. To multiply a and b, they are first converted to Montgomery form aR mod N and bR mod N.

$$\begin{aligned} \bar{a} &\equiv aR \bmod N \\ \bar{b} &\equiv \mathrm{b}R \bmod N \end{aligned} \tag{15}$$

Converting to and from Montgomery Form makes this slower than the conventional or *Barret* reduction algorithm for a single multiply. However, when performing many multiplications in a row, as in modular exponentiation, the intermediate result can be left in Montgomery form, and the initial and final conversions become a negligible fraction of the overall computation. Let R^{-1} and N' be integers satisfying $0 < \mathrm{R}^{-1} < \mathrm{N}$, $0 < \mathrm{N}' < \mathrm{R}$ and equation

$$RR^{-1} - NN' = 1 \tag{16}$$

Algorithm Reduction should be used to convert back from Montgomery form as shown in (17).

$$\begin{aligned} T &= \overline{ab} \\ m &= (T \bmod R)N' \bmod R \\ t &= \frac{T + mN}{R} \\ ab &= \begin{cases} t & (\text{if } t < N) \\ t - N & (\text{otherwise}) \end{cases} \end{aligned} \tag{17}$$

For comparison, the performance of Montgomery multiplication is 7 times faster than regular modular multiplication in fast modular exponentiation.

4. PROPOSED APPROACH

The general process of solving the problem is as follows:

1. Grid Calculation. It is done by applying the Shifting Evaluation Values algorithm recursively, which will be used for the next step.
2. Factorial Calculation. It is done by multiplying the result from Grid Calculation and adds the missing part of the multiplication which was not covered in the previous step.

This section is divided into several sections that explain the strategy of solving the SPOJ FACTMODP starting from Grid Calculation, Factorial Calculation, Optimization, and the Computational Approach.

4.1. Grid Calculation

The main idea in our approach is the grid calculation. This grid calculation is a recursive function, which has a base case and a recursive case. In this grid calculation step, we will interpolate the initial polynomial using Shifting Evaluation Value to get the desired result. Let N and X be integers that satisfying $\mathrm{X} = \mathrm{N}! \bmod \mathrm{P}$, $v = \lfloor \sqrt[2]{N} \rfloor$ and G_d is a polynomial with d terms. Defined a polynomial G_1 with the value of $\mathrm{G}_1(0) = 1$ and $\mathrm{G}_1(1) = \mathrm{v} + 1$ as the base case, this value will be used in the recursive case. The recursive case consists of 3 phases, namely shifting, combine, and extras phases.

1) *Shifting*: Let $\mathrm{d}' = \left\lfloor \frac{d}{2} \right\rfloor$, for every point d in 0 until d', we will do Shifting Evaluation Values on G_d three times, with different input of a,

$$
\begin{gathered}
G_{d'}(x) \xrightarrow{a=\frac{d'}{v}} G_{d'}\left(x + \frac{d'}{v}\right) \\
G_{d'}(x) \xrightarrow{a=d'} G_{d'}(x + d') \\
G_{d'}(x) \xrightarrow{a=\frac{d'v+d'}{v}} G_{d'}(x + \frac{d'v + d'}{v})
\end{gathered}
\tag{18}
$$

These Shifting Evaluation Values calculations are performed by Lagrange Interpolation and FFT/NTT. *Proof*: Defined a polynomial P and integers a and k, in Lagrange Interpolation, we have the following equation.

$$
\begin{aligned}
P(a + k) &= \sum_{i=0}^{d} P(i) \frac{\prod_{j=0,j\neq i}^{d}(a + k - j)}{\prod_{j=0,j\neq i}^{d}(i - j)} \\
&= \sum_{i=0}^{d} P(i) \cdot \prod_{j=0}^{d} a + k - j \cdot \frac{1}{a + k - i} \cdot \prod_{j=0,j\neq i}^{d} \frac{1}{i - j} \\
&= \left(\prod_{j=0}^{d} a + k - j\right) \cdot \left(\sum_{i=0}^{d} \frac{P(i)}{i!\,(d - i)!\,(-1)^{d-1}} \cdot \frac{1}{a + k - i}\right)
\end{aligned}
\tag{19}
$$

You will notice that the inside of the right parenthesis is in the form of convolution which represents the polynomial Q and the left parenthesis is $\Delta(a, i, d)$ in section 3.1. Now, we have the value of polynomial $G_{d'}(x)$ with $0 \leq x \leq 2d'$.

2) *Combine*: Given that shifted value is the multiplication of d' integers, to widen the multiplication range result returned in the previous phase, we should increase the d', in this phase, we will double the range. Let x is integers, for every point x in 0 until $2d'$,

$$
G_d(x) = G_{d'}(x) \,.\, G_{d'}\left(x + \frac{d'}{v}\right) \tag{20}
$$

3) *Extras*: In the case when $d \neq 2d'$ or when d is odd, we need to add extra multiplication so that the multiplication range size result returned in the shifting phase will equal to d and simplify the rest of the implementation. Let d is integers, so that

$$
\begin{gathered}
G_d(i) = G_d(i) \cdot (di + v), i \in [0, d) \\
G_{d(d)} = \prod_{i=1}^{d} dv + i.
\end{gathered}
\tag{21}
$$

After Grid Calculation is done, we will have the value of polynomial $G_v(i), i \in [0, v)$. As you may notice, the value of $G_v(i) = \prod_{j=1}^{v}(vi + j)$.

4.2. Factorial Calculation

Up to this point, we can rewrite equation (2) to equation (22). The process in the right parenthesis in equation (22) that's what we refer to as the missing part which was not covered in the Grid Calculation. The Grid Calculation was able to calculate factorial directly if N is a perfect square. The maximum complexity of calculating the missing part is $\mathcal{O}(2\sqrt{N})$.

$$N!\ mod\ P = \left(\prod_{i=0}^{v} G_v(i)\right) \cdot \left(\prod_{i=v^2}^{N} i\right) \bmod P \tag{22}$$

4.3. Optimization

In [2], Wilson's theorem is defined. Given a natural number $P > 1$, it is a prime number if and only if the product of all positive integers less than n is one less than a multiple of P. That is

$$P \text{ is prime } \Leftrightarrow (P-1)! \equiv -1 \pmod{P} \tag{23}$$

With this, we can cap out the number of multiplications to $P - N - 1$ and our complexity is based on P, not N, but we need additional steps to calculate inverse modulo from the result. Let $M = P - N - 1$ and $v = \lfloor \sqrt[2]{N} \rfloor$, we can rewrite the equation (22) to equation (24).

$$\begin{gathered} \text{Fact}(N) = \left(\prod_{i=0}^{v} G_v(d)\right) \cdot \left(\prod_{i=v^2}^{N} i\right) \\ N! \bmod P = \begin{cases} \text{Fact}(N) \bmod P, & (\text{if } 2N > P) \\ (-1)^M\ \text{Fact}(N)^{-1} \bmod P, & (\text{otherwise}) \end{cases} \end{gathered} \tag{23}$$

Since we are computing in the prime field, it would be better to do multiplication and division based on Montgomery Multiplication that has been discussed in section 3.5.

4.4. Computational Approach

This section will explain the computational approach to solve factorial. All pseudo-codes in this section will assume that every multiplication and division operation already implement Montgomery Multiplication inside it. To avoid repeating calculations, we will extract some of the components of Shifting Evaluation Values into several functions.

First, the Convert function showed in Algorithm 3, is a function that converts P to $\tilde{P}$ shown in equation (5). This function will be called in the Shifting Evaluation Values process. Second, the Shift function, showed in Algorithm 4 that represent the Shifting Evaluation Values process. Third, the Grid Calculation function showed in Algorithm 5 represents the Grid Calculation process. Finally, the Factorial function showed in Algorithm 6, is a function that wraps the whole process.

```
Algorithm 3 Convert Polynomial
Input: $f$
Output: $\bar{f}$
function CONVERT($f$)
    $n \leftarrow |f|$
    $ret \leftarrow f$
    for $i \leftarrow 0$ to $n - 1$ do
        $d \leftarrow i!^{-1} \times (n - 1 - i)!^{-1}$
        if $n - i - 1$ is odd then
            $d = -d$
        end if
        $ret[i] = ret[i] \times d$
    end for
    return $ret$
end function
```

Figure 3. Algorithm for Converting the Polynomial

Algorithm 4 Shift Polynomial

```
Input: $f, dx$
Output: Shifted $f$
function SHIFT($f, dx$)
    $n \leftarrow |f|$, $g \leftarrow []$
    $deg \leftarrow n - 1$
    $a \leftarrow \frac{dx}{\sqrt{N}}$
    $r \leftarrow a - deg$
    for $i \leftarrow 0$ to $2n - 1$ do
        $g[i] = g[i] \times \delta(i, d)$
    end for
    $ret \leftarrow$ MIDDLEPRODUCT($f, g$)
    for $i \leftarrow 0$ to $n - 1$ do
        $ret[i] = ret[i] \times \Delta(a, i, d)$
    end for
    return $ret$
end function
```

Figure 4. Algorithm for Shifting the Polynomial

Algorithm 5 Grid Calculation

```
Input: $f, dx$
Output: Value of $G_v$
function GRIDCALCULATION($n$)
    if $n$ is 1 then return $[1, 1 + \sqrt{N}]$
    end if
    $d' \leftarrow \lfloor \frac{n}{2} \rfloor$
    $g11 \leftarrow$ BOXCALCULATION($d'$)
    $g \leftarrow$ CONVERT($g11$)
    $g12 \leftarrow$ SHIFT($g, d'$)
    $g21 \leftarrow$ SHIFT($g, d' \times \sqrt{n}$)
    $g22 \leftarrow$ SHIFT($g, d' \times \sqrt{n} + d'$)
    for $i \leftarrow 0$ to $d'$ do
        $g11[i] = g11[i] \times g12[i]$
    end for
    for $i \leftarrow 1$ to $d'$ do
        $g11[i + d'] = g21[i] \times g22[i]$
    end for
    if $n$ is odd then
        for $i \leftarrow 0$ to $n - 1$ do
            $g11[i] = g11[i] \times \sqrt{N} \times i + n$
        end for
        $prod \leftarrow 1$
        for $i \leftarrow 1$ to $n$ do
            $prod = prod \times \sqrt{N} \times n + i$
        end for
        $g11.append(prod)$
    end if
    return $g11$
end function
```

Figure 5. Algorithm for Grid Calculation

As a note, the Middle Product function, called in Algorithm 4, is a function to calculate the middle product with modulo two prime numbers as in Table 1, then it will look for the real value using the Chinese Remainder Theorem as in the equation (11). To solve the FACTMODP problem, our solution requires 128-bit integers to save the temporary result of the multiplication inside the Montgomery World, this might be a little bit tricky, but there are several suggestions that you can implement, First, you can use the `__uint128_t` data type in C++ 64-bit. Second, building your data type, or you can integrate it with assembly language.

```
Algorithm 6 Factorial Calculation
Input: $n, p$
Output: $n! \bmod p$
 1: function FACTMODP($n, p$)
 2:     if $2n > p$ then
 3:         $m \leftarrow p - n - 1$
 4:         $ret \leftarrow$ FACTMODP($m, p$)
 5:         if $n$ is even then $ret \leftarrow -ret$
 6:         end if
 7:         return $ret^{-1}$
 8:     end if
 9:     SETMOD($p$)
10:     $f \leftarrow$ GRIDCALCULATION($n$)
11:     $partial \leftarrow \prod_{i=0}^{|f|} f[i]$
12:     $ret \leftarrow partial$
13:     $v \leftarrow \lfloor\sqrt{n}\rfloor$
14:     for $i \leftarrow 0$ to $n - v$ do
15:         $ret = ret \times (v + i + 1)$
16:     end for
17:     return $ret$
18: end function
```

Figure 6. Algorithm for Optimized Factorial Calculation

5. EVALUATION

In this section, we will evaluate the proposed approach. Two types of evaluation will be discussed in this section: the correctness test and the performance test. While the performance test is done by running the source code on a local computer to obtain the statistics of the program runtime which will then be compared accordingly, the correctness test will use the submission of source code in *Sphere Online Judge* (SPOJ) as a third-party online platform for source code checker. This source code is, then, executed and the result from the execution are compared to the answers provided by the problem maker.

5.1. Evaluation Methodology

On one hand, the correctness test is performed by submitting two source codes to SPOJ: Factorial Modulo Prime [6]. The first code is using our approach, while the second code is using Multipoint Evaluation adaptation. On other hand, the performance test is done by first generating the input by randomizing N and using fixed $P = 99999999977$ (largest prime number below 10^{11}). The test dataset is created by generating 50 random integers for each range $10^{I} - 10^{i+1}$, i $\in [0, 10]$. After being generated, this test dataset is then randomized and replicated to a 10-unit test for a single value of N, which mean there will be 5500 data inputs used for performance testing. We will test the dataset for every range of i. The performance test involves two main metrics which is runtime metric and memory metric.

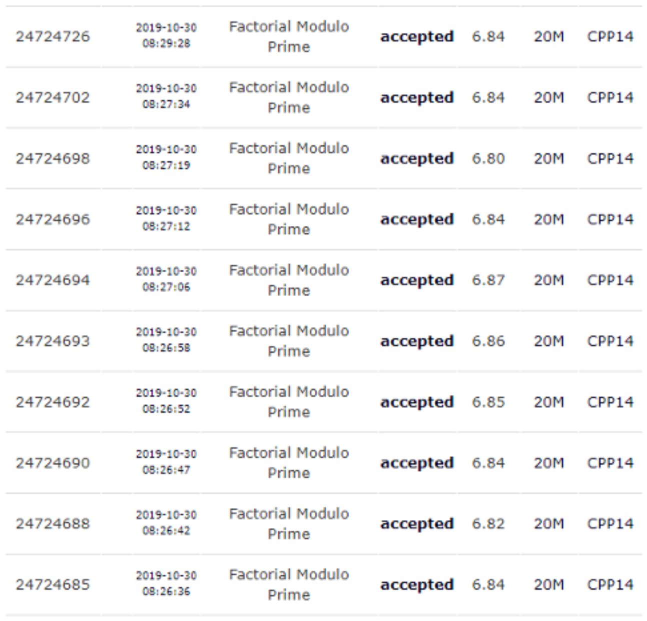
5.1. Correctness Test

Figure 7 and Figure 8 shows how the approaches used according to section 4 and [4] are scored in SPOJ. The code submissions have been executed at least 10 times to ensure the solution's both validity and consistency during the correctness test. Figure 9 shows how the solution is ranked among users in Sphere Online Judge. It is showed that our approach implementation takes the best solution to the first rank on the submission.

5.2. Performance Test

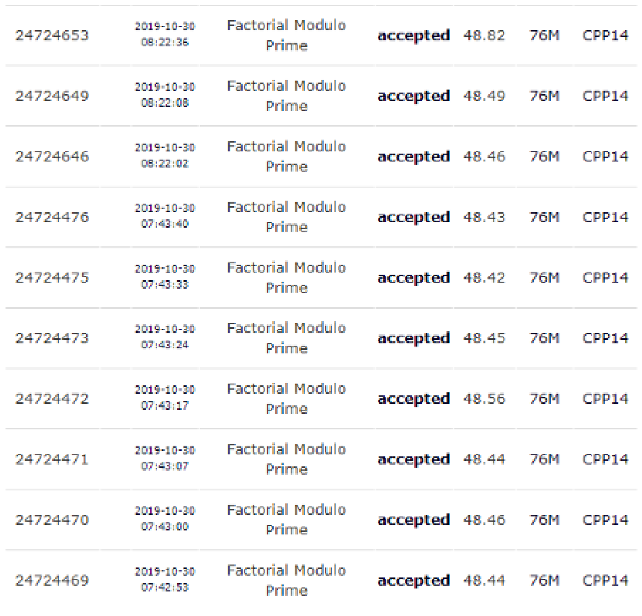
1) *Runtime*: Figure 10 shows a very distinct difference in the performance of each approach. The Multipoint Evaluation Adaptation approach has higher runtime due to its extra $\log N$ complexity. As in Figure 10, the runtime graph begins to decrease when $N \sim 5.10^{10}$ due to the implementation of Wilson's theorem, where the number of multiplications capped to $\frac{P}{2}$. Now, we will split Figure 10 into two plots, the first plot is the runtime where N is between $10^0 - 10^5$ (shown in Figure 11) and the second plot is the runtime where N is between $10^5 - 10^{11}$ (shown in Figure 12). As you can see, our approach has a slightly higher time than the Multipoint

Evaluation Adaptation approach in the small factorial, but our approach can run very quickly in the large factorial.



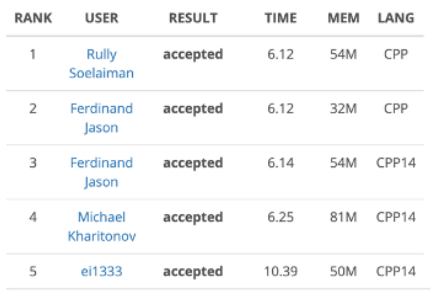
24724726	2019-10-30 08:29:28	Factorial Modulo Prime	**accepted**	6.84	20M	CPP14
24724702	2019-10-30 08:27:34	Factorial Modulo Prime	**accepted**	6.84	20M	CPP14
24724698	2019-10-30 08:27:19	Factorial Modulo Prime	**accepted**	6.80	20M	CPP14
24724696	2019-10-30 08:27:12	Factorial Modulo Prime	**accepted**	6.84	20M	CPP14
24724694	2019-10-30 08:27:06	Factorial Modulo Prime	**accepted**	6.87	20M	CPP14
24724693	2019-10-30 08:26:58	Factorial Modulo Prime	**accepted**	6.86	20M	CPP14
24724692	2019-10-30 08:26:52	Factorial Modulo Prime	**accepted**	6.85	20M	CPP14
24724690	2019-10-30 08:26:47	Factorial Modulo Prime	**accepted**	6.84	20M	CPP14
24724688	2019-10-30 08:26:42	Factorial Modulo Prime	**accepted**	6.82	20M	CPP14
24724685	2019-10-30 08:26:36	Factorial Modulo Prime	**accepted**	6.84	20M	CPP14

Figure 7. Correctness Test by using Our Approach



24724653	2019-10-30 08:22:36	Factorial Modulo Prime	**accepted**	48.82	76M	CPP14
24724649	2019-10-30 08:22:08	Factorial Modulo Prime	**accepted**	48.49	76M	CPP14
24724646	2019-10-30 08:22:02	Factorial Modulo Prime	**accepted**	48.46	76M	CPP14
24724476	2019-10-30 07:43:40	Factorial Modulo Prime	**accepted**	48.43	76M	CPP14
24724475	2019-10-30 07:43:33	Factorial Modulo Prime	**accepted**	48.42	76M	CPP14
24724473	2019-10-30 07:43:24	Factorial Modulo Prime	**accepted**	48.45	76M	CPP14
24724472	2019-10-30 07:43:17	Factorial Modulo Prime	**accepted**	48.56	76M	CPP14
24724471	2019-10-30 07:43:07	Factorial Modulo Prime	**accepted**	48.44	76M	CPP14
24724470	2019-10-30 07:43:00	Factorial Modulo Prime	**accepted**	48.46	76M	CPP14
24724469	2019-10-30 07:42:53	Factorial Modulo Prime	**accepted**	48.44	76M	CPP14

Figure 8. Correctness Test by using Multipoint Evaluation Adaptation



RANK	USER	RESULT	TIME	MEM	LANG
1	Rully Soelaiman	accepted	6.12	54M	CPP
2	Ferdinand Jason	accepted	6.12	32M	CPP
3	Ferdinand Jason	accepted	6.14	54M	CPP14
4	Michael Kharitonov	accepted	6.25	81M	CPP14
5	ei1333	accepted	10.39	50M	CPP14

Figure 9. Solution Ranking in SPOJ

Next, we will see from the standard deviation aspect as in Figure 13. In this figure, it is found that our approach seemingly remains stagnant at the standard deviation of 0.005 seconds. On the other hand, the Multipoint Evaluation Adaptation approach plot, this plot also approved that by the time N gets larger, the runtime of the program may get longer due to the standard deviation value of N starting from $N \sim 1.1 \,.\, 10^8$ reaches 0.01 seconds and going to increase. Also, there's a single spike every 10^i due to machine warm-up.

2) *Memory*: Figure 14 and 15 shows the maximum memory usage of each approach with N is between $10^0 - 10^6$ and $10^6 - 10^{11}$. Figure 14 states that our approach has a slightly higher memory usage than the Multipoint Evaluation Adaptation approach because we need to precompute inverse factorials. However, Figure 15 states that our approach consumes less memory than the Multipoint Evaluation Adaptation approach when calculating large factorial.

We can say that our approach has a better performance compared to the Multipoint Evaluation algorithm adaptation approach.

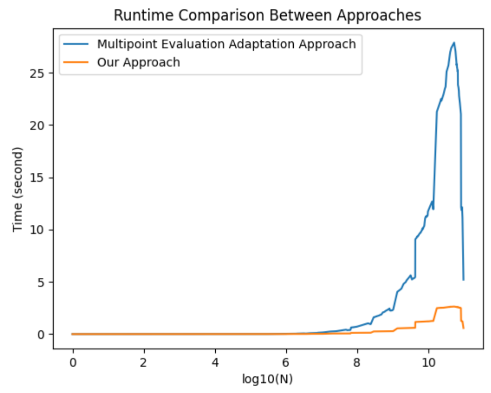


Figure 10. Runtime Comparison Between Approaches

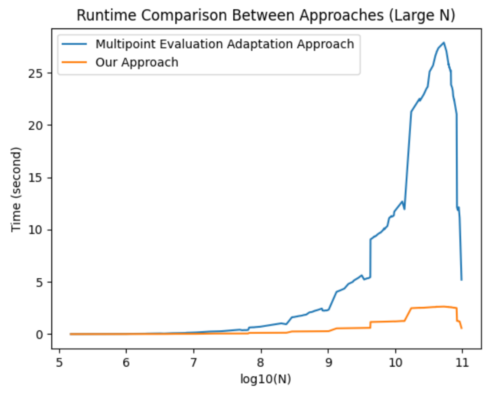


Figure 12. Runtime Comparison Between Approaches (Large N)

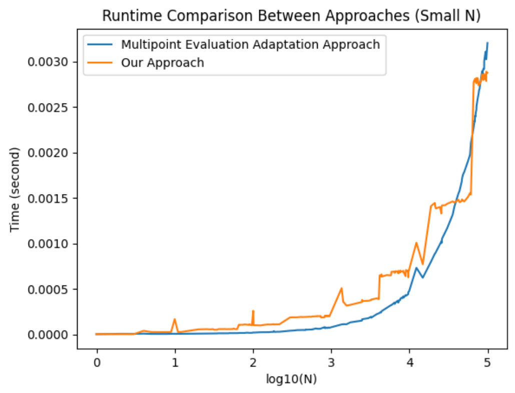


Figure 11. Runtime Comparison Between Approaches (Small N)

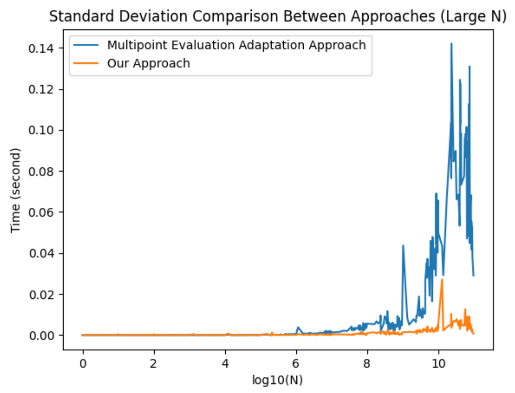


Figure 13. Standard Deviation Comparison Between Approaches

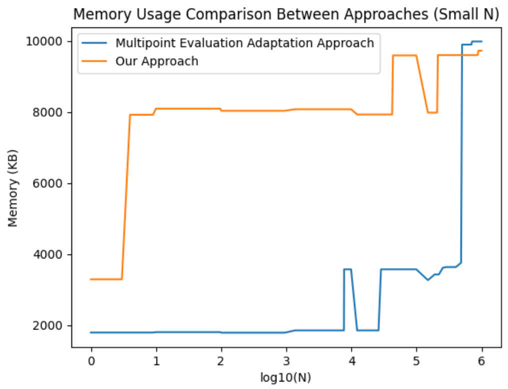


Figure 14. Memory Usage Comparison Between Approaches (Small N)

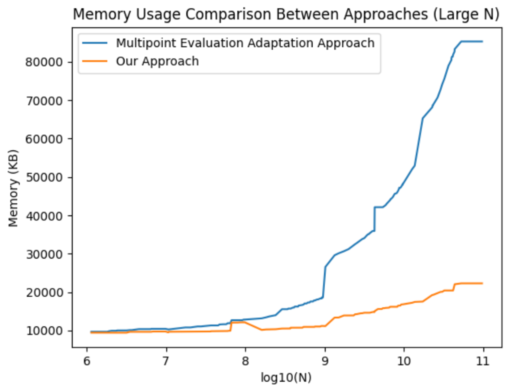


Figure 15. Memory Usage Comparison Between Approaches (Large N)

6. CONCLUSIONS

In this paper, we proposed a new algorithm to calculate factorials in the prime field efficiently. Our approach can run with complexity $O(\sqrt{n}\log\sqrt{n})$, as an example to calculate 53174223969! mod 99999999977 we need about 2.6-seconds. In the SPOJ Server, our approach has an average runtime of 6.84 seconds with a standard deviation of 0.0198 seconds and average maximum memory usage of 20 MB with a standard deviation of 0 MB, as a note, SPOJ will calculate the runtime and maximum memory usage for each test file.

This research is far from over since our approach has many restrictions. Further research is needed to provide a better approach for this problem, like selecting radix version used in NTT/FFT, integration with assembly languages, integration with quad-computing, integration with third-party computing libraries, selecting a base prime number for larger N or P, setting a threshold to use the efficient method to calculate the corresponding factorials, etc. Besides, further research is needed to apply this approach in factorial calculations that are not in the prime field or apply this approach in any sequence number calculations.

7. REFERENCES

[1] N. LBiggs, “The roots of combinatorics,” *Historia Mathematica*, vol. 6, pp. 109–136, 1979.
[2] T. Nagell, “Wilson’s theorem and its generalizations.” in *Introduction to Number Theory*. New York: Wiley, 1993, pp. 99–101, an optional note.
[3] R. J. Fateman, “Comments on factorial programs,” University of California, Berkeley, Tech. Rep., 4 2006.
[4] J. Gauthier, “Fast multipoint evaluation on n arbitrary points,” Simon Fraser University, Tech. Rep., 2017.
[5] J. v. z. Gathen and J. Gerhard, Modern Computer Algebra, 3rd ed. USA: Cambridge University Press, 2013.
[6] (2017)Spoj.[Online]. Available: https://www.spoj.com/problems/FACTMODP/
[7] A. Bostan, P. Gaudry, and ́E. Schost, “Linear recurrences with polynomial coefficients and computation of the Cartier-Manin operator on hyperelliptic curves,” in *Finite Fields and Applications* -Fq7, ser. LNCS, G. L. Mullen, A. Poli, and H. Stichtenoth, Eds., vol.2948. Toulouse, France: Springer Verlag, 2004, pp. 40–58. [Online]. Available: https://hal.inria.fr/inria-00514132
[8] G. Hanrot, M. Quercia, and P. Zimmermann, “The middle product algorithm i,” *Applicable Algebra in Engineering, Communication and Computing*, vol. 14, pp. 415–438, 2003.
[9] R. C. Agarwal and C. S. Burrus, “Number theoretic transforms to implement fast digital convolution,” *Proceedings of the IEEE*, vol. 63, no. 4, pp. 550–560, 1975.
[10] W. Stallings, Cryptography and Network Security: Principles and Practice, 6th ed. USA: Prentice Hall Press, 2013.
[11] H. Davenport and H. Montgomery, Multiplicative Number Theory, ser. Graduate Texts in Mathematics. Springer New York, 2013, p.124. [Online]. Available: https://books.google.co.id/books?id=SFztBwAAQBAJ
[12] P. Montgomery, “Modular multiplication without trial division,” *Mathematics of Computation*, vol. 44, pp. 519–521, 1985.