Fast Power Series Inversion: Newton's Iteration and the Middle Product Optimization

Hao Ze

Abstract

The normal way to divide two polynomials is to perform a classical polynomial long division, which requires a quadratic cost to compute the quotient and remainder. In this paper, we present a quasilinear power series inversion algorithm which is based on Newton's iteration and fast Fourier transforms. By employing this algorithm, we can determine the quotient and remainder in quasilinear time. We also utilize the middle product optimization which was first introduced by Hanrot, Querica and Zimmermann [2]. This improves the efficiency of our fast inversion algorithm by a constant factor. We implemented our algorithms in C and present some timings.

1 Introduction

Let $a(x)$ and $b(x)$ be two polynomials of degree n and m, respectively. When $n \geqslant m$, the classical polynomial long division requires $O((n-m+1)m)$ arithmetic operations to compute the quotient $q(x)$ and the remainder $r(x)$ where $r(x) = 0$ or $\deg(r(x)) < \deg(b(x))$. This method has a quadratic cost and hence is inefficient when n and m are large.

Our purpose is to develop an algorithm which performs polynomial at a quasilinear cost. We will show that this can be accomplished if we are able to determine the power series inverse of a given polynomial to some desired order n in $O(N \log N)$ operations where N is the first power of 2 greater than n. In order to achieve this, we exploit Newton's iteration and fast Fourier transforms. Then we develop two quasilinear power series inversion algorithms based on them. We finally utilize the middle product method given by Hanrot, Querica and Zimmermann [2] to optimize our inversion algorithm so that a power series inversion can always be performed in $10F(N)+O(N)$ where $F(N)$ is the cost of an FFT of size N, namely $F(N) \in O(N \log N)$.

We review the classical long division and introduce the concept of reciprocal polynomials in section 2. We explain Newton's iteration and develop a recursive power series inversion algorithm in section 3. In section 4, we expound the mechanism of the fast Fourier transform and prove the necessary theorems. In section 5, we present two quasilinear inversion algorithms and analyze the complexity. We introduce the middle product optimization in section 6 and prove its correctness. In the end, we do timings for our C implementations to verify the efficiency of our algorithms.

2 Division of Polynomials

Let F be a field. Given two polynomials $a(x)$ and $b(x)$ in $F[x]$ with $\deg(a(x)) = n$ and $\deg(b(x)) = m$, we want to compute $a(x)$ divided by $b(x)$. That is, we want to find the quotient $q(x)$ and the remainder $r(x)$ in $F[x]$ such that

$$a(x) = b(x)q(x) + r(x)$$

where $r(x) = 0$ or $\deg(r(x)) < \deg(b(x))$. Here, if $n < m$, then we just have $q(x) = 0$ and $r(x) = a(x)$. Thus, it is reasonable to assume $n \geqslant m$.

One way to achieve this is to perform the classical polynomial long division. We write $a(x), b(x)$ in descending order so that $a(x) = a_n x^n + a_{n-1}x^{n-1} + \cdots + a_1 x + a_0$ and $b(x) = b_m x^m + b_{m-1}x^{m-1} + \cdots + b_1 x + b_0$. Notice that $a_n, b_m \neq 0$. We do the division in the following steps.

1. We compute a_n/b_m in F to get the leading coefficient q_1 of $q(x)$.

$$\begin{array}{r} \\ b_m x^n + b_{m-1}x^{m-1} + \cdots + b_0 \end{array} \overline{\left) \begin{array}{l} q_1 x^{n-m} \\ \hline a_n x^n + a_{n-1}x^{n-1} + \cdots + a_{n-m}x^{n-m} + \cdots + a_1 x + a_0 \end{array} \right.}$$

   Then we multiply each term of $b(x)$ by $q_1 x^{n-m}$ to get a new polynomial whose leading term is also $a_n x^n$ and subtract this polynomial from $a(x)$. We let $a^{(1)}(x)$ denote the result and write it underneath.

$$\begin{array}{rl} & q_1 x^{n-m} \\ \hline b_m x^m + \cdots + b_0 \;\Big)\; & a_n x^n + a_{n-1}x^{n-1} + \cdots + a_{n-m}x^{n-m} + \cdots + a_1 x + a_0 \\ -(& a_n x^n + b_{m-1}q_1 x^{n-1} + \cdots + b_0 q_1 x^{n-m} \;) \\ \hline a^{(1)}(x) = & 0 + (a_{n-1} - b_{m-1}q_1)x^{n-1} + \cdots + (a_{n-m} - b_0 q_1)x^{n-m} + \cdots + a_0 \end{array}$$

2. If $a^{(1)}(x) \neq 0$ and $\deg(a^{(1)}(x)) = k \geqslant m$, then we find the leading coefficient ${a^{(1)}}_k$ of $a^{(1)}(x)$. We compute ${a^{(1)}}_k/b_m$ in F to get the second term $q_2 x^{k-m}$ of $q(x)$. Again, we multiply each term of $b(x)$ by $q_2 x^{k-m}$. We subtract this new product from $a^{(1)}(x)$ and use $a^{(2)}(x)$ to denote the resulting polynomial.

$$\begin{array}{rl} & q_1 x^{n-m} + q_2 x^{k-m} \\ \hline b_m x^m + \cdots + b_0 \;\Big)\; & a_n x^n + a_{n-1}x^{n-1} + \cdots + a_{n-m}x^{n-m} + \cdots + a_1 x + a_0 \\ -(& a_n x^n + b_{m-1}q_1 x^{n-1} + \cdots + b_0 q_1 x^{n-m} \;) \\ \hline & {a^{(1)}}_k x^k + {a^{(1)}}_{k-1}x^{k-1} + \cdots + a_1 x + a_0 \\ & -(\; {a^{(1)}}_k x^k + b_{m-1}q_2 x^{k-1} + \cdots + b_1 q_2 x^{k-m+1} + b_0 q_2 x^{k-m} \;) \\ \hline a^{(2)}(x) = & 0 + ({a^{(1)}}_{k-1} - b_{m-1}q_2)x^{k-1} + \cdots + a_1 x + a_0 \end{array}$$

3. We repeat step 2 to calculate $a^{(3)}(x), a^{(4)}(x), \ldots$ until $a^{(\ell)}(x) = 0$ or $\deg(a^{(\ell)}(x)) < n$. Then the polynomial on the top is $q(x)$ and $r(x) = a^{(\ell)}(x)$.

The following algorithm summarizes the procedure described above.

Algorithm Classical Polynomial Division

Input: $a, b \in F[x]$ where F is a field and $b \neq 0$.
Output: $q, r \in F[x]$ satisfying $a = b \cdot q + r$ where $r = 0$ or $\deg(r) < \deg(b)$.

$q \leftarrow 0$
$r \leftarrow a$
while $r \neq 0$ **and** $\deg(r) \geq \deg(b)$ **do**
  $\ell \leftarrow \text{LeadingTerm}(r)/\text{LeadingTerm}(b)$
  $q \leftarrow q + \ell$
  $r \leftarrow r - b \cdot \ell$
return q, r

Example 2.1. Let $a(x) = 5x^5 + 4x^4 - 2x^3 + x + 7$ and $b(x) = 2x^3 - 3x^2 - 5$. Suppose we want to divide $a(x)$ by $b(x)$ in $\mathbb{Z}_{13}[x]$. Performing the long division, we have

$$
\begin{array}{rl}
 & 9x^2 + 9x + 6 \\
\cline{2-2}
2x^3 - 3x^2 - 5 \;\big)\; & 5x^5 + 4x^4 - 2x^3 + x + 7 \\
 & -(\; 5x^5 + 12x^4 + 7x^2 \;) \\
\cline{2-2}
 & \quad 5x^4 - 2x^3 - 7x^2 + x + 7 \\
 & \quad -(\; 5x^4 - x^3 + 7x \;) \\
\cline{2-2}
 & \quad -x^3 - 7x^2 - 6x + 7 \\
 & \quad -(\; 12x^3 - 5x^2 - 4 \;) \\
\cline{2-2}
 & \quad -2x^2 - 6x + 11
\end{array}
$$

In this case, $q(x) = 9x^2 + 9x + 6$ and $r(x) = 11x^2 + 7x + 11$.

Now we consider the cost of the classical polynomial division by counting the arithmetic operations in F. During each step, we obtain one new term Δq of the quotient $q(x)$. Since $\deg(q(x)) = \deg(a(x)) - \deg(b(x)) = n - m$, $q(x)$ has at most $n - m + 1$ terms. Hence there are at most $n - m + 1$ steps needed. We compute Δq by one division in the field F. Then we multiply $b(x)$ by Δq. As $\deg(b(x)) = m$, $b(x)$ has at most $m + 1$ terms. But we already know the result of $b_m x^m \cdot \Delta q$, which is unnecessary to compute again. So we need at most m coefficient multiplications in F for each step. Similarly, when we do the

polynomial subtraction of each step, we know the difference between the leading terms must be 0, so we need at most m coefficient subtractions in F. Totally we have at most $(n-m+1)$ divisions, $(n-m+1)m$ multiplications and $(n-m+1)m$ subtractions in F. Thus, it requires $O((n-m+1)m)$ operations in F to compute $a(x)$ divided by $b(x)$.

We want to ask whether there is a faster way to compute the quotient and remainder. Notice that if $b(x)$ has a nonzero constant term, then $b(x)^{-1}$ exists and has a power series expansion in the ring $F[[x]]$. Suppose we can compute $b(x)^{-1}$. Since $a(x) = b(x)q(x) + r(x)$, we have

$$q(x) = a(x)b(x)^{-1} - r(x)b(x)^{-1}$$

One idea here is truncating the power series $b(x)^{-1}$ to the power of $n-m$. That is, we let $c(x) = b(x)^{-1} \mod x^{n-m+1}$. Then

$$q(x) = a(x)c(x) - r(x)c(x)$$

However, since $r(x)$ is unknown, we cannot compute $r(x)c(x)$ which might contain some term of degree less than $n-m+1$. So it seems we are unable to directly get $q(x)$ in this way.

In fact, we may accomplish our purpose by performing calculations on the reciprocal polynomials of the dividend and divisor.

Definition 2.1. Let $a(x) = a_n x^n + a_{n-1}x^{n-1} + \cdots + a_1 x + a_0$ with $a_n \neq 0$. The reciprocal polynomial of $a(x)$ is

$$a^r(x) = x^n a(1/x) = a_n + a_{n-1}x + \cdots + a_1 x^{n-1} + a_0 x^n.$$

Hence $a^r(x)$ is almost the same polynomial as $a(x)$ except its coefficients are in reversed order. Observe that

1. $a^r(x)^{-1}$ always exists in $F[[x]]$ since $a_n \neq 0$.
2. If $a_0 \neq 0$, $\deg(a^r(x)) = \deg(a(x))$. Otherwise, $\deg(a^r(x)) < \deg(a(x))$.

Another useful property of $a^r(x)$ is stated below.

Lemma 2.1. Let $a(x) = a_n x^n + a_{n-1}x^{n-1} + \cdots + a_1 x + a_0$ with $a_n \neq 0$. If $\deg(a^r(x)) = \ell$, then $a^{r^r}(x) = x^{\ell-n}a(x)$.

Proof. By definition, we have

$$a^{r^r}(x) = x^\ell a^r(\frac{1}{x}) = x^\ell \frac{1}{x^n} a(\frac{1}{1/x}) = x^{\ell-n}a(x).$$

□

This lemma implies $a^{r^r}(x) = a(x) \iff a_0 \neq 0$. As a result, we have the following theorem which provides an approach to our problem.

Theorem 2.2. Let F be a field. Suppose $a(x)$ and $b(x)$ are two polynomials in $F[x]$ with $\deg(a(x)) = n, \deg(b(x)) = m$ and $n \geqslant m$. Let $q(x)$ and $r(x)$ be the polynomials in $F[x]$ satisfying $a(x) = b(x)q(x) + r(x)$ where $r(x) = 0$ or $\deg(r(x)) < \deg(b(x))$. Then

$$q^r(x) = \frac{a^r(x)}{b^r(x)} \mod x^{n-m+1}.$$

Proof. Since $a(x) = b(x)q(x) + r(x)$, we have

$$a(1/x) = b(1/x)q(1/x) + r(1/x).$$

Multiplying both sides by x^n, we get

$$x^n a(1/x) = x^m b(1/x)x^{n-m}q(1/x) + x^n r(1/x).$$

As $\deg(q(x)) = \deg(a(x)) - \deg(b(x)) = n - m$, by definition, it follows that

$$a^r(x) = b^r(x)q^r(x) + x^n r(1/x). \tag{1}$$

If $r(x) = 0$, then $r(1/x) = 0$. Equation (1) becomes

$$a^r(x) = b^r(x)q^r(x).$$

Then the existence of $b^r(x)^{-1}$ implies

$$q^r(x) = \frac{a^r(x)}{b^r(x)}.$$

Since $\deg(q^r(x)) \leqslant \deg(q(x)) = n - m$, we obtain

$$q^r(x) = \frac{a^r(x)}{b^r(x)} \mod x^{n-m+1}.$$

Otherwise, when $r(x) \neq 0$, we let $d = \deg(r(x))$ and hence $0 \leqslant d \leqslant m - 1$. Then equation (1) becomes

$$\begin{aligned} a^r(x) &= b^r(x)q^r(x) + x^{n-d}x^d r(1/x) \\ a^r(x) &= b^r(x)q^r(x) + x^{n-d}r^r(x) \end{aligned}$$

Again, as $b^r(y)^{-1}$ exists, we get

$$q^r(x) = \frac{a^r(x)}{b^r(x)} - x^{n-d}\frac{r^r(x)}{b^r(x)}.$$

Now $d \leqslant m-1$ ensures $n-d \geqslant n-m+1$. $r^r(x)$ cannot be 0 due to $r(x) \neq 0$. We realize that the degree of any term in the power series $x^{n-d} r^r(x) b^r(x)^{-1}$ is at least $n-m+1$. Since $\deg(q^r(x)) \leqslant n-m$, it follows that

$$q^r(y) = \frac{a^r(y)}{b^r(y)} \mod y^{n-m+1}.$$

Therefore, in both cases of $r(x)$, the result is proved. □

At this point, we are able to compute the quotient $q(x)$ of $a(x)$ divided by $b(x)$ under the assumption that the inverse of $b^r(x)$ can be found in some way. We can find $q^r(x)$ by computing all the terms of degree less than $n-m+1$ in the product $a^r(x)b^r(x)^{-1}$. Once we get $q^r(x)$, we may easily obtain $q^{r^r}(x)$ by reversing the coefficients. If $q^{r^r}(x)$ and $q(x)$ have the same degree, then $q(x)$ is just $q^{r^r}(x)$. Otherwise, we multiply $q^{r^r}(x)$ by a power of x to make up the degree deficiency. Finally, we compute $r(x) = a(x) - b(x)q(x)$ to get the remainder.

The prerequisite here is that we need to know $b^r(x)^{-1}$, or at least $b^r(x)^{-1} \mod x^{n-m+1}$. For $b(x) = b_m x^m + b_{m-1}x^{m-1} + \cdots + b_0$ with $b_0 \neq 0$, one way to calculate $c(x) = b(x)^{-1}$ up to terms of degree k is using the classical power series inversion algorithm.

1. Write $(b(x) \mod x^{k+1})$ in ascending order. We compute $1/b_0$ in F to get the constant term c_0 of $c(x)$. Let $c^{(0)}(x) = c_0$. Then we multiply $(b(x) \mod x^{k+1})$ by c_0 and subtract this product from 1. Let $e^{(0)}(x)$ denote the result.

$$\begin{array}{rl} & c_0 \\ \hline b_0 + b_1 x + \cdots + b_k x^k \;\big)\; & 1 \\ & -(\; 1 + b_1 c_0 x + \cdots + b_k c_0 x^k \;) \\ \hline e^{(0)}(x) = & 0 - b_1 c_0 x - \cdots - b_k c_0 x^k \end{array}$$

2. If $k > \deg(c^{(0)}(x)) = 0$ and $e^{(0)}(x) \neq 0$, we find the term in $e^{(0)}(x)$ which has the lowest degree, say ${e^{(0)}}_t x^t$. We compute ${e^{(0)}}_t / b_0$ in F to get the coefficient c_t of $c(x)$, and let $c^{(1)}(x) = c^{(0)}(x) + c_t x^t$. Then we multiply $(b(x) \mod x^{k-t+1})$ by $c_t x^t$ to get another polynomial of degree k. We subtract it from $e^{(0)}(x)$ and let $e^{(1)}(x)$ denote the resulting polynomial.

$$\begin{array}{rl} & c_0 + c_t x^t \\ \hline b_0 + b_1 x + \cdots + b_{k-t} x^{k-t} \;\big)\; & 1 \\ & -(\; 1 + b_1 c_0 x + \cdots + b_k c_0 x^k \;) \\ \hline & {e^0}_t x^t + {e^0}_{t+1} x^{t+1} + \cdots - b_k c_0 x^k \\ & -(\; {e^0}_t x^t + b_1 c_t x^{t+1} + \cdots + b_{k-t} c_t x^k \;) \\ \hline e^{(1)}(x) = & 0 + ({e^0}_{t+1} - b_1 c_t) x^{t+1} + \cdots - (b_k c_0 + b_{k-t} c_t) x^k \end{array}$$

3. We repeat step 2 to update $c^{(\ell)}(x)$ by one term each time until $\deg(c^{(\ell)}(x)) = k$ or $e^{(\ell)}(x) = 0$. Then $c(x) = c^{(\ell)}(x) = b(x)^{-1} \mod x^{k+1}$.

Algorithm Classical Power Series Inversion

Input: $b \in F[x]$ where F is a field and $b_0 \neq 0$, a positive integer k.
Output: $c \in F[x]$ satisfying $c = b^{-1} \mod x^{k+1}$.

ClassicalInversion(b, k):
  $e \leftarrow 1$
  $c \leftarrow 0$
  $t \leftarrow 0$
  while $e \neq 0$ **and** $t < k$ **do**
    $\ell \leftarrow$ the term of the lowest degree in e
    $\ell \leftarrow \ell / b_0$
    $t \leftarrow \deg(\ell)$
    $c \leftarrow c + \ell$
    $e \leftarrow e - (b \mod x^{k+1-t}) \cdot \ell$
  return c

This algorithm requires at most $k + 1$ steps to determine $c(x) = (b(x)^{-1} \mod x^{k+1})$ since $c(x)$ is updated by one new term each time and $\deg(c(x)) \leqslant k$. At each step i, we need one coefficient division in F to compute the new term $c_i x^i$ of $c(x)$. We only need to multiply $b_1 x + b_2 x^2 + \cdots + b_{k-i} x^{k-i}$ by $c_i x^i$ because any term of degree higher than k is useless to our calculation and it is unnecessary to compute $b_0 \cdot c_i x^i$. Hence we need at most $k - i$ multiplications in F. For the polynomial subtraction, we know $b_0 c_i x^i$ must cancel out the term of the lowest degree $e^{(i-1)}(x)$, so we need at most $k - i$ subtractions in F. Totally, we need at most $(k + 1)$ divisions, at most $\sum_{i=0}^{k}(k - i) = \frac{1}{2}(k^2 + k)$ multiplications and subtractions. It thus takes $O(k^2)$ operations in F to compute $b(x)^{-1}$ truncated to degree k.

Combining this inversion algorithm with the results achieved from Lemma 2.1 and Theorem 2.2, we obtain an alternative way to perform polynomial divisions.

Algorithm Polynomial Division by Power Series Inversion

Input: $a, b \in F[x]$ where F is a field and $b \neq 0$.

Output: $q, r \in F[x]$ satisfying $a = b \cdot q + r$ where $r = 0$ or $\deg(r) < \deg(b)$.

$k \leftarrow \deg(a) - \deg(b)$ ▷ $k = \deg(q)$

if $k < 0$ **then return** $q = 0, r = a$

if $\deg(b) = 0$ **then return** $q = a/b_0, r = 0$

$c \leftarrow \text{ClassicalInversion}(b^r, k)$

$d \leftarrow a^r \cdot c \mod x^{k+1}$

$\ell \leftarrow \deg(d)$

$q \leftarrow x^{k-\ell} d^r$

$r \leftarrow a - b \cdot q$

return q, r

Example 2.2. For $a(x) = x^5 - 3x^4 - x^3 + 2x^2 + x - 1$ and $b(x) = 2x^3 - 3x^2 + 8$, we want to divide $a(x)$ by $b(x)$ in $\mathbb{Z}_{13}[x]$.

We have $a^r(x) = 1 - 3x - x^2 + 2x^3 + x^4 - x^5$ and $b^r(x) = 2 - 3x + 8x^3$. As $\deg(q(x))$ must be 2, we need to determine the terms of degree at most 2 in $a^r(x)b^r(x)^{-1}$. Notice $a^r(x)$ has a nonzero constant term, so we need to compute $b^r(x)^{-1}$ up to terms of degree 2. We perform $\text{ClassicalInversion}(b^r(x), 1)$.

$$
\begin{array}{rl}
 & 7 + 4x + 6x^2 \\
\cline{2-2}
2 - 3x \;\big) & 1 \\
 & -(\;1 + 5x\;) \\
\cline{2-2}
 & 8x \\
 & -(\;8x + x^2\;) \\
\cline{2-2}
 & 12x^2
\end{array}
$$

Hence $b^r(x)^{-1} \mod x^3 = 7 + 4x + 6x^2$. Then we get

$$q^r(x) = (1 - 3x - x^2 + 2x^3 + x^4 - x^5)(7 + 4x + 6x^2) \mod x^3 = 7 + 9x.$$

So $q(x) = x \cdot q^{r^r}(x) = x(7x + 9) = 7x^2 + 9x$,

$$r(x) = (x^5 - 3x^4 - x^3 + 2x^2 + x - 1) - (2x^3 - 3x^2 + 8)(7x^2 + 9x) = 11x^2 + 7x + 12.$$

Now we turn to the complexity of our new division algorithm. We perform $a(x) = \sum_{i=0}^{n} a_i x^i$ divided by $b(x) = \sum_{i=0}^{m} b_i x^i$ with $n \geqslant m$ by determining $(b^r(x)^{-1} \mod x^{n-m+1})$, $q^r(x)$, $q(x)$ and $r(x)$ in turn.

1. As a consequence of the analysis on page 7, we are able to find $(b^r(x)^{-1} \mod x^{n-m+1})$ at a cost of $O((n-m)^2)$.

2. Computing $q^r(x) = a^r(x)b^r(x)^{-1} \mod x^{n-m+1}$ requires one multiplication of $(a^r(x) \mod x^{n-m+1})$ and $(b^r(x)^{-1} \mod x^{n-m+1})$. Let $M(n)$ denote the cost of multiplying two polynomials of degree less than n. Then such a multiplication costs $M(n-m+1)$, which depends on the multiplication method we choose. For instance, if we apply the classical polynomial multiplication to expand this product and subsequently truncate the result, the cost will be $(n-m+1)^2$. However, by noticing the calculations of all the terms of degree higher than $n-m$ can be omitted in advance, we have a cheaper way to do the expansion. Writing $b^r(x)^{-1} \mod x^{n-m+1} = c_0 + c_1x + \cdots + c_{n-m-1}x^{n-m-1} + c_{n-m}x^{n-m}$, we obtain

$$\begin{aligned} & (a_n + a_{n-1}x + \cdots + a_m x^{n-m})(c_0 + c_1 x + \cdots + c_{n-m}x^{n-m}) \mod x^{n-m+1} \\ = \; & a_n(c_0 + c_1x + \cdots + c_{n-m}x^{n-m}) + a_{n-1}x(c_0 + c_1x + \cdots + c_{n-m-1}x^{n-m-1}) \\ & + \cdots + a_m x^{n-m}c_0 \end{aligned} \tag{2}$$

   We need $\sum_{i=1}^{n-m+1} i = \frac{1}{2}(n-m+1)(n-m+2)$ multiplications and additions in F. Hence the cost of computing $q^r(x)$ in this way is $\frac{1}{2}(n-m+1)^2 + O(n-m)$.

3. Reversing the coefficients of $q^r(x)$ requires a linear cost. When $\deg(q^r(x)) = \ell < n-m$, we need increase the degree of every term in $q^{r^r}(x)$ by $n-m-\ell$ to get $q(x)$. This process is also linear. So it costs $O(n-m)$ to determine $q(x)$.

4. Computing $r(x) = a(x) - b(x)q(x)$ requires one polynomial multiplication and one polynomial subtraction. The subtraction cost is linear, and the multiplication cost here depends on the greater one of $m+1$ and $n-m+1$. So we can obtain $r(x)$ at a cost of $M(\max\{m+1, n-m+1\}) + O(n+1)$.

Consequently, the total cost of this algorithm is

$$\begin{aligned} & O((n-m)^2) + M(n-m+1) + O(n-m) + M(\max\{m, n-m\}+1) + O(n+1) \\ =& O((n-m)^2) + O(M(\max\{m+1, n-m+1\})). \end{aligned}$$

We see that the efficiency of the algorithm is mainly determined by the inversion and multiplication methods we employ. In practice, we can reduce $M(n)$ to $O(n^{\log_2 3})$ by Karatsuba's algorithm or to $O(n \log n)$ by the fast Fourier transform, which will be discussed later in this paper. Moreover, we are able to save the cost of a power series inversion by using Newton's iteration.

3 Newton's Iteration

Newton's iteration is an iterative algorithm for approximating the solution to a nonlinear equation $f(x) = 0$. The iteration starts with an initial guess of the root near the correct answer, say x_0. Given $f(x)$ is differentiable and $f'(x_0) \neq 0$, the Taylor series for $f(x)$ at $x = x_0$ is

$$f(x) = f(x_0) + f'(x_0)(x - x_0) + O((x - x_0)^2).$$

Let $T(x) = f(x_0) + f'(x_0)(x - x_0) \approx f(x)$. Solving the linear equation $T(x_1) = 0$ for x_1, we get

$$x_1 = x_0 - \frac{f(x_0)}{f'(x_0)}.$$

Then x_1 becomes the next approximation to the solution. Repeating the process, we obtain $x_2, x_3, x_4, \ldots$ where $x_{k+1} = x_k - f(x_k)/f'(x_k)$. With a good initial guess x_0, the iteration generates a sequence $\{x_k\}_{k=0}^{\infty}$ which converges to the correct root of the equation $f(x) = 0$. Furthermore, if that root is a simple root, the rate of convergence is quadratic. See page 137 of Geddes, Czapor and Labahn [1] and page 70 of Burden and Faires [4] for more details about the convergence of Newton's iteration.

Definition 3.1. We say a power series $\bar{p}(x)$ is an order n approximation of $p(x)$ if $\bar{p}(x) = p(x) + O(x^n) = p(x) \mod x^n$.

Then for some polynomial $a(x) = \sum_{i=0}^{m} a_i x^i$ with $a_0 \neq 0$, Newton's iteration provides a way to generate a sequence of polynomials that approximates the power series $a(x)^{-1}$ to higher and higher order.

Let $y(x) = a(x)^{-1}$. Finding $y(x)$ is equivalent to solving the equation

$$a(x) - \frac{1}{y(x)} = 0.$$

Define $f(y) = a(x) - 1/y$. Then solving $f(y_k) + f'(y_k)(y_{k+1} - y_k) = 0$ for y_{k+1} gives

$$\begin{aligned} y_{k+1} &= y_k - \frac{f(y_k)}{f'(y_k)} \\ &= y_k - \frac{a(x) - 1/y_k}{1/{y_k}^2} \\ &= y_k + y_k(1 - y_k \cdot a(x)). \end{aligned}$$

Suppose y_k is an order ℓ approximation of $y(x)$, which means $y_k = y(x) + O(x^\ell)$. It then follows that

$$\begin{aligned} y_{k+1} &= 2y(x) + 2O(x^\ell) - \Big[y(x)^2 + 2y(x)O(x^\ell) + O(x^\ell)^2\Big]a(x) \\ &= 2y(x) + 2O(x^\ell) - y(x) - 2O(x^\ell) - O(x^\ell)^2 a(x) \\ &= y(x) - O(x^\ell)^2 a(x) \end{aligned}$$

$$= y(x) - O(x^{2\ell})a(x)$$

Since $a(x) \neq 0$, we have $y_{k+1} = a(x)^{-1} + O(x^{2\ell})$. Thus, y_{k+1} is an order 2ℓ approximation of $a(x)^{-1}$. This means we can double the correct terms of $a(x)^{-1}$ after a new iteration. Hence the accuracy of our approximation is doubled each time. The sequence $\{y_0, y_1, y_2, \dots\}$ converges to $a(x)^{-1}$ quadratically.

Accordingly, we are able to find an order n approximation of $y(x)$ in $\lceil n/2 \rceil$ iterations. Moreover, we notice that

$$f'(\frac{1}{a_0}) = \frac{1}{1/{a_0}^2} = {a_0}^2 \neq 0,$$

so we may choose $y_0 = 1/a_0$ as our initial approximation. Since $f(y_0) = a(x) - a_0 = a_1x + a_2x^2 + a_3x^3 + \cdots = 0 \mod x$, y_0 is an order 1 approximation of $y(x)$. Then by induction on k, it is clear that $y_k = y(x) + O(x^{2^k})$. Therefore, the formula for the k^{th} iteration is

$$y_{k+1} = y_k + y_k(1 - y_k \cdot a(x)) \mod x^{2^{k+1}}.$$

But in our polynomial division problem, if the divisor is a constant, then the quotient and remainder can be immediately determined, namely $q(x) = a(x)/b_0$ and $r(x) = 0$, at a linear cost. Hence we may reasonably assume that the input polynomial of the following power series inversion algorithm is not a constant.

Algorithm Newton's Iteration for Power Series Inversion

Input: A polynomial $a(x)$ with $a_0 \neq 0$ and a positive integer n.
Output: An order n approximation of $a(x)^{-1}$.

$y \leftarrow 1/a_0$
$N \leftarrow 1$
while $N < n$ **do**
  $N \leftarrow 2N$
  $b \leftarrow a(x) \mod x^N$
  $y \leftarrow y + y(1 - y \cdot b) \mod x^N$
return $y \mod x^n$

We illustrate this iteration algorithm by the following example.

Example 3.1. Let $a(x) = 3 - 4x + 7x^2 + 9x^4 - 5x^5 + 2x^6 + x^7 + O(x^8) \in \mathbb{Z}_{11}[[x]]$. Setting $y_0 = 1/3 = 4$, we can compute an order 5 approximation of $a(x)^{-1}$ with 3 iterations.

$$\begin{aligned}
y_1 &= 4 + 4(1 - 4 \cdot (3 - 4x)) \mod x^2 \\
&= 4 + 9x \\
y_2 &= 4 + 9x + (4 + 9x)(1 - (4 + 9x)(3 - 4x + 7x^2)) \mod x^4 \\
&= 4 + 9x + 10x^2 + 7x^3 \\
y_3 &= (4 + 9x + 10x^2 + 7x^3) + (4 + 9x + 10x^2 + 7x^3)(1 - (4 + 9x + 10x^2 + 7x^3)
\end{aligned}$$

$$(3 - 4x + 7x^2 + 9x^4 - 5x^5 + 2x^6 + x^7)) \mod x^8$$
$$= 4 + 9x + 10x^2 + 7x^3 + 7x^4 + 2x^5 + 9x^6 + 3x^7$$

Then $y(x) = 4 + 9x + 10x^2 + 7x^3 + 7x^4$ is an order 5 approximation of $a(x)^{-1}$. As we expect, $y(x) \cdot a(x) = 1 + 5x^5 + 3x^6 + 2x^7 + \cdots = 1 + O(x^5)$.

In the above example, we see a flaw. From y_2 to $y(x)$, we only need one new term, but we computed y_3 which contains twice as many terms as y_2 has. In fact, we can modify our algorithm to get a recursive version that saves some work.

Algorithm Recursive Newton Inversion Algorithm

Input: A polynomial $a(x)$ with $a_0 \neq 0$ and a positive integer n.
Output: An order n approximation of $a(x)^{-1}$.

RNI$(a(x), n)$:
  if $n = 1$ **then return** $y \leftarrow 1/a_0$
  $m \leftarrow \lceil n/2 \rceil$
  $y \leftarrow \text{RNI}((a(x), m)$
  $b \leftarrow a(x) \mod x^n$
  $\left.\begin{array}{l} \sum_{i=0}^{n-1} g_i x^i \leftarrow y \cdot b \mod x^n \\ f \leftarrow \sum_{i=0}^{n-m-1} (-g_{i+m}) x^i \\ \sum_{i=0}^{n-m-1} h_i x^i \leftarrow y \cdot f \mod x^{n-m} \\ y \leftarrow y + \sum_{i=0}^{n-m-1} h_i x^{i+m} \end{array}\right\} y \leftarrow y + y(1 - y \cdot b) \mod x^n$
  return y

The algorithm RNI$(a(x), n)$ indeed makes use of the same mechanism as the previous one does. They both need $\lceil \log_2 n \rceil$ steps. Nevertheless, the sizes of the polynomials y, b required at each step are reduced. For instance, suppose we are given some power series $a(x)$ with $a_0 \neq 0$ and we want to find an order 18 approximation of $a(x)^{-1}$. Instead of doing calculations modulo x^2, x^4, x^8, x^{16} and x^{32} in turn, we may recursively compute better and better approximations modulo x^2, x^3, x^5, x^9 and x^{18}.

In addition, RNI$(a(x), n)$ gives a detailed procedure which shows how we may compute $(y + y(1 - y \cdot b) \mod x^n)$ with less cost. First, since b has degree at most $n - 1$ and y has degree at most $m-1$ where $m = \lceil n/2 \rceil$, it costs $M(n)$ to compute $(y \cdot b \mod x^n)$. Intuitively, we might think computing $(y(1 - y \cdot b) \mod x^n)$ requires $M(n)$ as well because $(1 - y \cdot b \mod x^n)$ has degree at most $n - 1$. However, as we know $y = a(x)^{-1} \mod x^m$ and $b = a(x) \mod x^n$, we must have

$$y \cdot b \mod x^n = 1 + 0 \cdot x + 0 \cdot x^2 + 0 \cdot x^3 + \cdots + 0 \cdot x^{m-1} + g_m \cdot x^m + g_{m+1} \cdot x^{m+1} + \cdots + g_{n-1} \cdot x^{n-1}$$

for some coefficients g_i's in F. So it follows that

$$1 - y \cdot b \mod x^n = -g_m \cdot x^m - g_{m+1} \cdot x^{m+1} - \cdots - g_{n-2} \cdot x^{n-2} - g_{n-1} \cdot x^{n-1}$$

$$= x^m(-g_m - g_{m+1} \cdot x - \cdots - g_{n-2} \cdot x^{n-m-2} - g_{n-1} \cdot x^{n-m-1})$$

Let f denote $(-g_m - g_{m+1} \cdot x - \cdots - g_{n-2} \cdot x^{n-m-2} - g_{n-1} \cdot x^{n-m-1})$. We may simplify our calculation of $(y(1 - y \cdot b) \mod x^n)$ by computing $h = (y \cdot f \mod x^{n-m})$ first. As $\deg(f) \leqslant n - m - 1 = n - \lceil n/2 \rceil - 1 \leqslant \lceil n/2 \rceil - 1$, both y and f have degree at most $\lceil n/2 \rceil - 1$. In consequence, h can be obtained by one multiplication of cost $M(\lceil n/2 \rceil)$. Then we increase the degree of every term in h by m and add the resulting polynomial to y. This has a linear cost cn for some constant c. Therefore, we are able to compute $(y + y(1 - y \cdot b) \mod x^n)$ in $M(n) + M(\lceil n/2 \rceil) + cn$ rather than $2M(n) + cn$.

Now we may consider the total complexity of $\mathrm{RNI}(a(x), n)$. Let $I(n)$ denote the cost of performing $\mathrm{RNI}(a(x), n)$ for some power series $a(x) \in F[[x]]$. Based on a reasonable assumption that $M(2n) > 2M(n)$ for all $n \geqslant 1$, the following theorem provides an upper bound for $I(n)$.

Theorem 3.1. Let F be a field and $a(x)$ be a power series in $F[[x]]$ with $a_0 \neq 0$. If $N = 2^k$ is the first power of 2 greater than $n-1$, namely $\frac{N}{2} < n \leqslant N$, and assuming $M(2n) > 2M(n)$ for all $n \geqslant 1$, then $I(n) \in O(M(N))$:

$$I(n) < 3M(N) + O(N).$$

Proof. Clearly, $I(1) = 1$ since we only need one division in F to obtain $1/a_0$. For $n \geqslant 2$, we first find an order $\lceil n/2 \rceil$ approximation y by a recursive call $\mathrm{RNI}(a(x), \lceil n/2 \rceil)$ at a cost of $I(\lceil n/2 \rceil)$. Then we compute $y + y(1 - y \cdot b) \mod x^n$. According to the discussion above, it costs $M(n) + M(\lceil n/2 \rceil) + cn$ to perform this computation. Hence we obtain the recurrence relation

$$I(n) = I(\lceil n/2 \rceil) + M(n) + M(\lceil n/2 \rceil) + cn, \qquad I(1) = 1.$$

This gives

$$\begin{aligned} I(n) \leqslant I(N) &= I(\frac{N}{2}) + M(N) + M(\frac{N}{2}) + cN \\ &= I(\frac{N}{4}) + M(\frac{N}{2}) + M(\frac{N}{4}) + c(\frac{N}{2}) + M(N) + M(\frac{N}{2}) + cN \\ &\quad \vdots \\ &= I(1) + M(2) + M(4) + \cdots + M(\frac{N}{2}) + M(N) + M(1) + M(2) + \cdots + \\ &\quad M(\frac{N}{4}) + M(\frac{N}{2}) + c(2 + 4 + \cdots + N) \end{aligned}$$

As $N = 2^k$ and $M(n) < 2^{-1}M(2n)$, it follows that $M(1) < 2^{-1}M(2) < 2^{-2}M(4) < \cdots < 2^{-k}M(N)$. Then we have

$$\begin{aligned} I(N) < I(1) &+ 2^{1-k}M(N) + 2^{2-k}M(N) + \cdots + 2^{-1}M(N) + M(N) \\ &+ 2^{-k}M(N) + 2^{1-k}M(N) + \cdots + 2^{-2}M(N) + 2^{-1}M(N) + c(2 + 2^2 + \cdots + 2^k) \end{aligned}$$

$$
\begin{aligned}
&= 1 + 2^{1-k}(1+2^{-1})M(N) + 2^{2-k}(1+2^{-1})M(N) + \cdots + 2^{-1}(1+2^{-1})M(N) \\
&\quad + (1+2^{-1})M(N) + c(2^{k+1}-2) \\
&= 1 + \frac{3}{2}M(N)\sum_{j=1}^{k} 2^{1-j} + c(2N-2) \\
&< \frac{3}{2}M(N)\sum_{j=1}^{\infty} 2^{1-j} + 2cN - 2c + 1 \\
&= \frac{3}{2}\cdot 2M(N) + 2cN - 2c + 1 \\
&= 3M(N) + O(N)
\end{aligned}
$$

We finally derive an upper bound for the complexity of $\mathrm{RNI}(a(x), n)$:

$$I(n) \leqslant I(N) < 3M(N) + O(N)$$

where $N/2 < n \leqslant N = 2^k$ for some k. Thus, $I(n) \in O(M(N))$ since $M(N) \notin O(N)$. □

If the classical polynomial multiplication is employed, then computing $(y + y(1 - y \cdot b) \bmod x^n)$ will have a quadratic cost in $O(n^2)$. More precisely, it will consume $n\lceil n/2 \rceil$ to expand $y \cdot b$ and $(\lceil n/2 \rceil)^2$ to expand $y \cdot f$. For simplicity, we assume n is a power of 2. Then we will have

$$I(n) = I(\frac{n}{2}) + \frac{3}{4}n^2 + cn.$$

It follows that

$$
\begin{aligned}
I(n) &= I(1) + \frac{3}{4}(n^2 + \frac{n^2}{4} + \frac{n^2}{16} + \cdots + 4) + c(n + \frac{n}{2} + \frac{n}{4} + \cdots + 2) \\
&< 1 + \frac{3}{4}n^2 \sum_{j=0}^{\infty} 4^{-j} + c(2n-2) \\
&= \frac{4}{3}\cdot\frac{3}{4}n^2 + 2cn - 2c + 1 \\
&= n^2 + O(n)
\end{aligned}
$$

One obvious drawback here is that we compute many unnecessary terms during these two expansions. Since we will truncate our results after the expansions, the calculations of those terms, which will be eliminated by the truncation, are superfluous and hence should be avoided. This is similar to the idea shown in equation (2) of section 2. In general, given two polynomials $a(x) = \sum_{i=0}^{s-1} a_i x^i$ and $b(x) = \sum_{i=0}^{t-1} b_i x^i$ with $a_{s-1}, b_{t-1} \neq 0$, we have

$$\sum_{i=0}^{s-1} a_i x^i \sum_{i=0}^{t-1} b_i x^i \mod x^k = \sum_{i=0}^{k-1}\sum_{j=0}^{i} a_j b_{i-j} x^i \tag{3}$$

for all k satisfying $1 \leqslant k \leqslant min\{s,t\}$. The total number of coefficient multiplications needed is $\sum_{i=0}^{k-1}(i+1) = \frac{1}{2}k(k+1)$. Accordingly, the cost of expanding $(y \cdot f \mod x^{n/2})$ can be reduced to $\frac{1}{2}(n/2)(n/2+1) = n^2/8 + n/4$. The case for $(y \cdot b \mod x^n)$ is a bit more complicated. The premise of equation (3) is not met because $\deg(y)+1 < n$. Consequently, we cannot directly apply equation (3). Instead, by writing $y = z_0 + z_1 x + \cdots + z_{n/2-1}x^{n/2-1}$ and $b = b_0 + b_1 x + \cdots + b_{n-1}x^{n-1}$, we see that

$$\begin{aligned} y \cdot b \mod x^n = {} & y(b_0 + b_1 x + \cdots + b_{n/2}x^{n/2}) + (z_0 + z_1 x + \cdots + z_{n/2-2}x^{n/2-2})b_{n/2+1}x^{n/2+1} \\ & + (z_0 + z_1 x + \cdots + z_{n/2-3}x^{n/2-3})b_{n/2+2}x^{n/2+2} + \cdots + (z_0 + z_1 x)b_{n-2}x^{n-2} \\ & + z_0 b_{n-1}x^{n-1} \end{aligned}$$

The cost of expanding $y(b_0 + b_1 x + \cdots + b_{n/2}x^{n/2})$ is $n/2 \cdot (n/2+1) = n^2/4 + n/2$. The rest costs $\sum_{i=1}^{n/2-1} i = \frac{1}{2}(n/2-1)(n/2) = n^2/8 - n/4$. So the total cost of the multiplications at an order n recursive step drops to

$$\frac{n^2}{8} + \frac{n}{4} + \frac{n^2}{4} + \frac{n}{2} + \frac{n^2}{8} - \frac{n}{4} = \frac{n^2}{2} + \frac{n}{2}.$$

Therefore, we obtain another recurrence

$$I(n) = I(\frac{n}{2}) + \frac{n^2}{2} + cn.$$

We solve it in the same manner.

$$\begin{aligned} I(n) &= I(1) + \frac{1}{2}(n^2 + \frac{n^2}{4} + \frac{n^2}{16} + \cdots + 4) + c(n + \frac{n}{2} + \frac{n}{4} + \cdots + 2) \\ &< \frac{n^2}{2}\sum_{j=0}^{\infty} 4^{-j} + 2cn - 2c + 1 \\ &= \frac{2n^2}{3} + O(n) \end{aligned}$$

In comparison with the classical polynomial multiplication, we see

$$\frac{2n^2/3 + O(n)}{n^2 + O(n)} \sim \frac{2}{3}.$$

Although the improved multiplication method still has a unsatisfactory quadratic complexity, it is about 1/3 faster than the classical one.

Now we exhibit an implementation of our power series inversion algorithm. In order to express polynomials appropriately in programming language, we consider an array representation of polynomials. Given a polynomial $a(x)$ of degree m, we can express $a(x)$ as an array A by storing the coefficient a_j at position j of A.

$$A = [a_0, a_1, a_2, \ldots, a_{m-1}, a_m, 0, 0, 0, \ldots]$$

Obviously, A should have length at least $m+1$ to completely represent $a(x)$.

The function `modinv(c,p)` computes the inverse of c in $\mathbb{Z}_p$ by using the extended Euclidean algorithm.

```
int modinv( int c, int p )
{   int d, r, q, r1, c1, d1;
    d = p; c1 = 1; d1 = 0;
    while( d!=0 ) {
       q = c/d;
       r = c-q*d; r1 = c1-q*d1;
       c = d; c1 = d1;
       d = r; d1 = r1; }
    if( c!=1 ) return( 0 );
    if( c1<0 ) c1 += p;
    return( c1 );
}
```

The function `polmul(A,B,C,da,db,d,p)` applys the improved polynomial multiplication method to compute $C = A \cdot B \mod x^{d+1}$ in $\mathbb{Z}_p[x]$, where $\deg(A) = da$ and $\deg(B) = db$. The polynomials A, B and C are all in array representations. We adopt the positive representation for integers modulo p.

```
int max( int a, int b ) { if(a>b) return a; else return b; }
int min( int a, int b ) { if(a<b) return a; else return b; }

    void polmul( int *A, int *B, int *C, int da, int db, int d, int p )
{   int i, k; LONG t, M;
    M = ((LONG) p)*p;
    for( k=min(d,da+db); k>=0; k-- ) {
        t = M;
        for( i=max(0,k-db); i<=min(k,da); i++ )
           { if( t<0 ) t += M; t -= ((LONG) A[i])*((LONG) B[k-i]); }
        t = t%p; t = -t; if( t<0 ) t += p;
        C[k] = t; }
    return;
}
```

The function `NewtonSlow(a,n,T,y,p)` performs $\mathrm{RNI}(a, n)$ based on `polmulB()` to compute $y = a^{-1} \mod x^n$ for a given $a \in \mathbb{Z}_p[x]$. The input array T is used as temporary storage for intermediate calculations. Its length should be at least n.

```
void NewtonSlow( int *a, int n, int *T, int *y, int p )
{   if( n==1 ) { y[0] = modinv(a[0], p); return; }
    int m = (n+1)/2;
    NewtonSlow( a, m, T, y, p );
    polmul( y, a, T, m-1, n-1, n-1, p );
    for( int i=m; i<=n-1; i++ )
        if( T[i]!=0 ) T[i] = p-T[i];
    polmul( y, T+m, y+m, m-1, n-m-1, n-m-1, p );
    return;
}
```

As we have shown above, the efficiency of our power series inversion algorithm highly depends on the multiplication cost $M(n)$. The two methods we discussed are both quadratic, which leads the complexity $I(n)$ to be $O(n^2)$ as well. Hence we seek some other multiplication algorithm that is faster than $O(n^2)$. One possible choice is Karatsuba's algorithm. It takes $O(n^{\log_2 3}) \approx O(n^{1.585})$ arithmetic operations in F to multiply two polynomials in $F[x]$. See pages 118-119 of Geddes, Czapor and Labahn [1] for details. In fact, there is an even faster method. We can perform polynomial multiplications in quasilinear time by utilizing fast Fourier transforms.

4 Fast Fourier Transform

A fast Fourier transform (FFT) evaluates or interpolates a polynomial in quasilinear time on the basis of the symmetry property of the Fourier points. So we begin the discussion with an introduction of Fourier points.

Definition 4.1. In a field F, an element ω is a primitive n-th root of unity if

$$\omega^n = 1, \quad \text{but } \omega^k \neq 1 \text{ for } 0 < k < n.$$

Then the set of n points $\{1, \omega, \omega^2, \ldots, \omega^{n-1}\}$ are called Fourier points.

Lemma 4.1. If ω is a primitive n-th root of unity with n even, then ω^2 is a primitive $n/2$-th root of unity.

Proof. ω^2 is an $n/2$-th root of unity because

$$(\omega^2)^{n/2} = \omega^n = 1.$$

In order to show ω^2 is also primitive, we assume there exists some $0 < k < n/2$ such that $(\omega^2)^k = 1$. Then we have

$$\omega^{2k} = 1 \quad \text{with } 0 < 2k < n.$$

This contradicts that ω is a primitive n-th root of unity. Thus, ω^2 is a primitive $n/2$-th root of unity. □

Example 4.1. In $\mathbb{Z}_{17}$, 8 is a primitive 8th root of unity since $8^8 = 1 \bmod 17$ and

$$8^2 = -4, \quad 8^3 = 2, \quad 8^4 = -1, \quad 8^5 = -8, \quad 8^6 = 4, \quad 8^7 = -2. \pmod{17}$$

The corresponding 8 Fourier points are $\{1, 8, -4, 2, -1, -8, 4, -2\}$.
Moreover, 8^2 is a primitive 4th root of unity. The corresponding 4 Fourier points are $\{1, 8^2, 8^4, 8^6\} = \{1, -4, -1, 4\}$.
And 8^4 is a primitive 2nd root of unity. The corresponding 2 Fourier points are $\{1, 8^4\} = \{1, -1\}$.

In the above example, we observe that $8^{4+i} = -8^i$ for $i \in \{0, 1, 2, 3\}$. In reality, such a symmetry exists in every set of Fourier points and is crucial to an FFT. The following lemma gives an explicit statement of this property.

Lemma 4.2. If ω is a primitive n-th root of unity, then the n Fourier points satisfy

$$\omega^{n/2+i} = -\omega^i$$

for $i \in \{0, 1, \ldots, n/2 - 1\}$.

Proof. Since $\omega^n = 1$, we have

$$\begin{aligned}(\omega^{n/2+i} + \omega^i)(\omega^{n/2+i} - \omega^i) &= (\omega^{n/2+i})^2 - (\omega^i)^2 \\ &= \omega^n(\omega^i)^2 - (\omega^i)^2 \\ &= 0.\end{aligned}$$

If $\omega^{n/2+i} - \omega^i = 0$, then $\omega^{n/2+i} = \omega^i$. We will get $\omega^{n/2} = 1$, which contradicts that ω is primitive. Therefore, $\omega^{n/2+i} + \omega^i = 0$. □

Now we start to expound the mechanism of fast Fourier transforms.

Definition 4.2. Let F be a field and ω be a primitive n-th root of unity in F. The discrete Fourier transform (DFT) is a mapping $T_\omega : F^n \to F^n$ defined by

$$T_\omega(a_0, a_1, a_2, \ldots, a_{n-1}) = (\hat{a}_0, \hat{a}_1, \hat{a}_2, \ldots, \hat{a}_{n-1}),$$

where

$$\hat{a}_i = a_0 + a_1\omega^i + a_2(\omega^i)^2 + \cdots + a_{n-1}(\omega^i)^{n-1}.$$

We realize that the DFT is a linear transformation from the vector space F^n to itself. Its matrix in the standard basis is given by

$$\begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & \omega & \omega^2 & \cdots & \omega^{n-1} \\ 1 & \omega^2 & \omega^4 & \cdots & \omega^{2(n-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega^{n-1} & \omega^{2(n-1)} & \cdots & \omega^{(n-1)(n-1)} \end{bmatrix}$$

As a consequence, the DFT is a homomorphism which preserves the addition and multiplication of F^n. In other words, for any

$$(f_0, f_1, \ldots, f_{n-1}), (g_0, g_1, \ldots, g_{n-1}) \in F^n,$$

we have the following properties:

(i) $T_\omega(f_0, f_1, \ldots, f_{n-1}) + T_\omega(g_0, g_1, \ldots, g_{n-1}) = T_\omega(f_0 + g_0, f_1 + g_1, \ldots, f_{n-1} + g_{n-1})$

(ii) $T_\omega(f_0, f_1, \ldots, f_{n-1}) \cdot T_\omega(g_0, g_1, \ldots, g_{n-1}) = T_\omega(f_0 \cdot g_0, f_1 \cdot g_1, \ldots, f_{n-1} \cdot g_{n-1})$

From Definition 4.2, we also see that if $a(x) = a_0 + a_1x + a_2x^2 + \cdots + a_{n-1}x^{n-1}$, then

$$T_\omega(a_0, a_1, a_2, \ldots, a_{n-1}) = (a(1), a(\omega), a(\omega^2), \ldots, a(\omega^{n-1})).$$

Hence, for any polynomial $f(x) \in F[x]$ with $\deg(f(x)) \leqslant n-1$, the DFT can evaluate $f(x)$ at $\{1, \omega, \omega^2, \ldots, \omega^{n-1}\}$. Normally, it requires $O(n^2)$ arithmetic operations in F to evaluate a polynomial of degree $n-1$ at n points. Nevertheless, by taking advantage of the symmetry property of Fourier points, we are able to perform such evaluations with a quasilinear cost.

Suppose $a(x)$ is a polynomial of degree $n-1$ in $F[x]$. We wish to evaluate $a(x)$ by the DFT. Let N be the first power of 2 greater than $n-1$, and ω be a primitive N-th root of unity in F. Again, when $n < N$, we add dummy zero terms to $a(x)$ so as to make the number of terms be N. As a result, we get $a(x) = \sum_{i=0}^{N-1} a_i x^i$. Then our goal is to compute $T_\omega(a_0, a_1, a_2, \ldots, a_{N-1})$. If $N = 1$, then $a(x) = a_0$ is a constant. Accordingly, $T_1(a_0) = (\hat{a}_0) = (a_0)$. Otherwise, we rearrange the terms of $a(x)$ in the form

$$a(x) = (a_0 + a_2x^2 + a_4x^4 + \cdots + a_{N-2}x^{N-2}) + (a_1 + a_3x^3 + a_5x^5 + \cdots + a_{N-1}x^{N-1}).$$

Then we can rewrite $a(x)$ as

$$a(x) = b(x^2) + x \cdot c(x^2) \tag{4}$$

where

$$b(y) = a_0 + a_2y + a_4y^2 + \cdots + a_{N-2}y^{N/2-1} \quad \text{and} \quad c(y) = a_1 + a_3y + a_5y^2 + \cdots + a_{N-1}y^{N/2-1}.$$

In other words, $b(x^2)$ is the sum of the even degree terms of $a(x)$ and $c(x^2)$ is the sum of the odd degree terms of $a(x)$ with the power of x reduced by 1. By definition, we should determine

$$\hat{a}_i = a_0 + a_1\omega^i + a_2(\omega^i)^2 + \cdots + a_{N-1}(\omega^i)^{N-1} = a(\omega^i)$$

for $i \in \{0, 1, 2, \ldots, N-1\}$. Equation (4) implies that we may compute

$$a(\omega^i) = b(\omega^{2i}) + \omega^i \cdot c(\omega^{2i})$$

for $i \in \{0, 1, 2, \ldots, N-1\}$. Thanks to the symmetry property revealed by Lemma 4.2, we have

$$\omega^{N/2+i} = -\omega^i, \quad \text{and hence } \omega^{2(N/2+i)} = \omega^{2i}.$$

In consequence, we only need to compute $b(\omega^{2i})$ and $c(\omega^{2i})$ from which we get

$$a(\omega^i) = b(\omega^{2i}) + \omega^i \cdot c(\omega^{2i}), \qquad a(\omega^{N/2+i}) = b(\omega^{2i}) - \omega^i \cdot c(\omega^{2i}) \tag{5}$$

for $i \in \{0, 1, 2, \ldots, N/2-1\}$. In addition, we notice that both $b(y)$ and $c(y)$ have degree at most $N/2-1$. By Lemma 4.1, ω^2 is a primitive $N/2$-th root of unity with the set of $N/2$ Fourier points $\{1, \omega^2, \omega^4, \ldots, \omega^{N-2}\}$. Thus, we are capable of evaluating $b(y)$ and $c(y)$ at these $N/2$ points by another two DFTs, $T_{\omega^2}(a_0, a_2, a_4, \ldots, a_{N-2})$ and $T_{\omega^2}(a_1, a_3, a_5, \ldots, a_{N-1})$, respectively. After evaluating $b(y)$ and $c(y)$, we combine the results into the desired evaluations of $a(x)$ by formula (5).

Now we present the fast Fourier transform algorithm in full detail.

Algorithm Fast Fourier Transform

Input: $a(x)$ is a polynomial in $F[x]$ where F is a field,
N is a power of 2 such that $N/2 < \deg(a(x)) + 1 \leqslant N$,
$W = (W_0, W_1, W_2, \ldots, W_{N/2-1}) = (1, \omega, \omega^2, \ldots, \omega^{N/2-1})$ where ω is a primitive N-th root of unity in F.

Output: $(a(1), a(\omega), a(\omega^2), \ldots, a(\omega^{n-1}))$.

FFT$(a(x), N, W)$:
  if $N = 1$ **then return** $A_0 \leftarrow a_0$
  $b(x) \leftarrow \sum_{i=0}^{N/2-1} a_{2i} x^i$ $\qquad$ $c(x) \leftarrow \sum_{i=0}^{N/2-1} a_{2i+1} x^i$
  $(B_0, B_1, \ldots, B_{N/2-1}) \leftarrow \text{FFT}(b(x), N/2, (W_0, W_2, W_4, \ldots, W_{N/2-2}))$
  $(C_0, C_1, \ldots, C_{N/2-1}) \leftarrow \text{FFT}(c(x), N/2, (W_0, W_2, W_4, \ldots, W_{N/2-2}))$
  for i from 0 to $N/2 - 1$ **do**
    $A_i \leftarrow B_i + W_i \cdot C_i$
    $A_{N/2+i} \leftarrow B_i - W_i \cdot C_i$
  return $(A_0, A_1, \ldots, A_{N-1})$

Example 4.2. In Example 4.1, we find that -4 is a primitive 4th root of unity in $\mathbb{Z}_{17}$ and the corresponding 4 Fourier points are $\{1, -4, -1, 4\}$. Say we want to evaluate

$$a(x) = 3x^3 + 6x^2 - x + 5$$

at these 4 points. We write $a(x) = b(y) + x \cdot c(y)$ where $y = x^2$ and

$$b(y) = 6y + 5, \qquad c(y) = 3y - 1.$$

We first need to evaluate $b(y)$ and $c(y)$ at $\{1, -1\}$. Again, we write $b(y) = d(z) + x \cdot e(z)$ and $c(y) = f(z) + x \cdot g(z)$ where $z = y^2$ and

$$d(z) = 5, \qquad e(z) = 6, \qquad f(z) = -1, \qquad g(z) = 3.$$

Since

$$d(1) = 5, \qquad e(1) = 6, \qquad f(1) = -1, \qquad g(1) = 3,$$

we have

$$b(1) = d(1) + e(1) = 11, \qquad b(-1) = d(1) - e(1) = -1,$$
$$c(1) = f(1) + g(1) = 2, \qquad c(-1) = f(1) - g(1) = -4.$$

Combining these evaluations, we obtain

$$a(1) = b(1) + c(1) = -4, \qquad a(-1) = b(1) - c(1) = -8,$$
$$a(-4) = b(-1) + (-4) \cdot c(-1) = -2, \qquad a(4) = b(-1) - (-4) \cdot c(-1) = 0.$$

Thus, the final result is $T_{-4}(5, -1, 6, 3) = (-4, -2, -8, 0)$.

Theorem 4.3. Let N be a power of 2 and ω be a primitive N-th root of unity. The corresponding Fourier points are precomputed and stored in W, namely $W = (1, \omega, \omega^2, \ldots, \omega^{N/2-1})$. If $F(N)$ denotes the cost of performing $\mathrm{FFT}(a(x), N, W)$ for a polynomial $a(x)$ with $\deg(a(x)) < N$, then

$$F(N) \in O(N \log N).$$

Proof. When $N = 1$, we directly return the constant a_0 without any operation. Clearly, $F(1) = 0$.
When $N \geqslant 2$, we need two recursive calls to evaluate $b(x)$ and $c(x)$ at $N/2$ points. Since $b(x)$ and $c(x)$ both have degree at most $N/2 - 1$, each call requires $F(N/2)$ operations. In addition, for $0 \leqslant i \leqslant N/2 - 1$, since each ω^i is precomputed, we need $N/2$ multiplications to determine $w^i C_i$'s, $N/2$ additions to determine A_i's and $N/2$ subtractions to determine $A_{N/2+i}$'s. Therefore, we obtain the recurrence relation

$$F(1) = 0, \qquad F(N) = 2F(\frac{N}{2}) + c\frac{N}{2}$$

for some constant c. With $N = 2^k$, we have

$$\begin{aligned} F(N) &= 2(2F(\frac{N}{4}) + c\frac{N}{4}) + c\frac{N}{2} \\ &= 2^2 F(\frac{N}{4}) + 2c\frac{N}{2} \\ &= 2^3 F(\frac{N}{8}) + 3c\frac{N}{2} \\ &\vdots \\ &= 2^k F(1) + kc\frac{N}{2} \\ &= \frac{c}{2} kN \end{aligned}$$

As $k = \log N$, it follows that

$$F(N) = \frac{c}{2} N \log N.$$

Furthermore, if $\{\omega^2, \omega^3, \ldots, \omega^{N/2-1}\}$ are not given along with ω, then they should be calculated beforehand. This will require an extra $N/2 - 2$ multiplications. Accordingly, the total cost will become $\frac{c}{2} N \log N + O(N)$. Thus, the complexity of the FFT algorithm is $O(N \log N)$. □

After we obtain the evaluations of a polynomial at a set of Fourier points, we seek an efficient way to do polynomial interpolation. That is, we wish to reconstruct the correct polynomial from the results of the DFT. One possible way is to invert the associated matrix of the linear transformation T_ω by using Gaussian elimination. But this will require $O(n^3)$ operations. Other polynomial interpolation methods such as Lagrange interpolation

and Newton interpolation can be performed in $O(n^2)$ operations (see Chapter 5 of Geddes, Czapor and Labahn [1]), which is still unsatisfactory. We want an even faster method, say a quasilinear one. Actually, this can be accomplished by using another FFT.

Definition 4.3. Let F be a field and ω be a primitive n-th root of unity in F. The inverse discrete Fourier transform (IDFT) is a mapping $S_\omega : F^n \to F^n$ defined by

$$S_\omega(b_0, b_1, b_2, \ldots, b_{n-1}) = (\bar{b}_0, \bar{b}_1, \bar{b}_2, \ldots, \bar{b}_{n-1}),$$

where

$$\bar{b}_j = n^{-1}(b_0 + b_1\omega^{-j} + b_2(\omega^{-j})^2 + \cdots + b_{n-1}(\omega^{-j})^{n-1}).$$

We claim that the IDFT is the inverse of the DFT.

Theorem 4.4. Let F be a field and ω be a primitive n-th root of unity in F. For any $(a_0, a_1, a_2, \ldots, a_{n-1}) \in F^n$, if

$$T_\omega(a_0, a_1, a_2, \ldots, a_{n-1}) = (\hat{a}_0, \hat{a}_1, \hat{a}_2, \ldots, \hat{a}_{n-1}),$$

then

$$S_\omega(\hat{a}_0, \hat{a}_1, \hat{a}_2, \ldots, \hat{a}_{n-1}) = (a_0, a_1, a_2, \ldots, a_{n-1}).$$

Proof. Since ω is a primitive n-th root of unity, we have

$$(\omega^p)^n = (\omega^n)^p = 1, \quad \text{and } \omega^p \neq 1,$$

for any integer power p satisfying $0 < p < n$. The identity

$$x^n - 1 = (x-1)(x^{n-1} + x^{n-2} + \cdots + x + 1)$$

implies that

$$(\omega^p - 1)((\omega^p)^{n-1} + (\omega^p)^{n-2} + \cdots + \omega^p + 1) = (\omega^p)^n - 1 = 0.$$

As $\omega^p - 1 \neq 0$, we must have

$$(\omega^p)^{n-1} + (\omega^p)^{n-2} + \cdots + \omega^p + 1 = 0.$$

Also, if we multiply both sides by $(\omega^{-p})^{n-1}$, we get

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

where $-n < -p < 0$. Combining the above two equations, we obtain

$$(\omega^p)^{n-1} + (\omega^p)^{n-2} + \cdots + \omega^p + 1 = 0, \qquad \text{whenever } 0 < |p| < n. \tag{6}$$

Now we are ready to show $S_\omega = {T_\omega}^{-1}$.
First, by Definition 4.2, $T_\omega(a_0, a_1, a_2, \ldots, a_{n-1}) = (\hat{a}_0, \hat{a}_1, \hat{a}_2, \ldots, \hat{a}_{n-1})$ means

$$\hat{a}_i = \sum_{k=0}^{n-1} a_k(\omega^i)^k, \quad \text{for } i = 0, 1, \ldots, n-1.$$

And by Definition 4.3, the j-th component of $S_\omega(\hat{a}_0, \hat{a}_1, \hat{a}_2, \ldots, \hat{a}_{n-1})$ is in the form

$$n^{-1} \sum_{i=0}^{n-1} \hat{a}_i(\omega^{-j})^i.$$

Then for any $j \in \{0, 1, \ldots, n-1\}$, we have

$$\begin{aligned} n^{-1} \sum_{i=0}^{n-1} \hat{a}_i \cdot \omega^{-ij} &= n^{-1} \sum_{i=0}^{n-1} \sum_{k=0}^{n-1} a_k \cdot \omega^{ik} \cdot \omega^{-ij} \\ &= n^{-1} \sum_{k=0}^{n-1} a_k \sum_{i=0}^{n-1} \omega^{i(k-j)}. \end{aligned}$$

The result of the inside summation $\sum_{i=0}^{n-1} \omega^{i(k-j)}$ depends on whether j and k are equal. When $k = j$, we get

$$\sum_{i=0}^{n-1} \omega^{i(k-j)} = \sum_{i=0}^{n-1} \omega^0 = \sum_{i=0}^{n-1} 1 = n.$$

When $k \neq j$, we have $0 < |k - j| < n$ since $j, k \in \{0, 1, \ldots, n-1\}$. Then

$$\sum_{i=0}^{n-1} \omega^{i(k-j)} = (\omega^{k-j})^{n-1} + (\omega^{k-j})^{n-2} + \cdots + \omega^{k-j} + 1 = 0$$

by equation (6). It follows that

$$n^{-1} \sum_{k=0}^{n-1} a_k \sum_{i=0}^{n-1} \omega^{i(k-j)} = n^{-1} \cdot a_j \cdot n = a_j,$$

which means the j-th component of $S_\omega(\hat{a}_0, \hat{a}_1, \hat{a}_2, \ldots, \hat{a}_{n-1})$ is exactly a_j.
Consequently, we obtain

$$S_\omega(\hat{a}_0, \hat{a}_1, \hat{a}_2, \ldots, \hat{a}_{n-1}) = (a_0, a_1, a_2, \ldots, a_{n-1}).$$

□

Now we return to our original aim. The following algorithm exploits fast Fourier transforms to perform polynomial multiplications in quasilinear time.

Algorithm Polynomial Multiplications by Fast Fourier Transforms

Input: Two polynomials $a(x), b(x)$ in $F[x]$ where F is a field.
Output: $c(x) = a(x) \cdot b(x)$

FFTMul$(a(x), b(x))$:
  $N \leftarrow$ the first power of 2 greater than $\deg(a(x)) + \deg(b(x))$
  $\omega \leftarrow$ a primitive N-th root of unity in F
  $W \leftarrow (1, \omega, \omega^2, \ldots, \omega^{N/2-1})$
  $(A_0, A_1, \ldots, A_{N-1}) \leftarrow \mathrm{FFT}(a(x), N, W)$
  $(B_0, B_1, \ldots, B_{N-1}) \leftarrow \mathrm{FFT}(b(x), N, W)$
  for i from 0 to $N-1$ **do** $C_i = A_i \cdot B_i$
  $V \leftarrow (1, \omega^{-1}, \omega^{-2}, \ldots, \omega^{-(N/2-1)})$
  $(c_0, c_1, \ldots, c_{N-1}) \leftarrow N^{-1} \cdot \mathrm{FFT}(\sum_{i=0}^{N-1} C_i x^i, N, V)$
  $c(x) \leftarrow \sum_{i=0}^{N-1} c_i x^i$
  return $c(x)$

The correctness of this algorithm follows from Theorem 4.4 and the fact that the DFT is a homomorphism. Notice that we need $N > \deg(a(x)) + \deg(b(x)) = \deg(c(x))$ so as to have an adequate number of Fourier points to interpolate $c(x)$. In order to determine the complexity, we count the number of arithmetic operations required. There are three FFTs of size N. According to Theorem 4.3, they cost a total of $3F(N)$ operations where $F(N) \in O(N \log N)$. The Fourier points can be computed iteratively. For instance, $\omega^2 = \omega \cdot \omega, \omega^3 = \omega^2 \cdot \omega, \omega^4 = \omega^3 \cdot \omega$, etc. So it takes $N/2 - 2$ field multiplication to obtain W. One optimization here is that we can save the additional computations of $(\omega^{-1}, \omega^{-2}, \ldots, \omega^{-(N/2-1)})$ by negating the values of $(\omega^{N/2-1}, \omega^{N/2-2}, \ldots, \omega)$ in turn. Notice that

$$\omega^{-i} = \omega^{n-i} = -\omega^{n/2-i}$$

since $\omega^n = 1$ and $\omega^{n/2+i} = -\omega^i$. As a consequence, we have

$$\{1, \omega^{-1}, \omega^{-2}, \ldots, \omega^{-(n/2-1)}\} = \{1, -\omega^{n/2-1}, -\omega^{n/2-2}, \ldots, -\omega\}.$$

Hence V can be obtained simply based on W. Lastly, it takes N field multiplications to determine C_i's for $0 \leqslant i \leqslant N-1$. Combining these costs, we see the total complexity here is

$$3F(N) + O(N)$$

where N is a power of 2 and $N/2 < \deg(a(x)) + \deg(b(x)) < N$. Therefore, we obtain a $O(N \log N)$ algorithm for polynomial multiplications.

Example 4.3. Suppose we want to compute

$$a(x) \cdot b(x) - c(x)^2 + 5$$

in $\mathbb{Z}_{17}[x]$, where

$$a(x) = 5x^3 + 4x + 1, \qquad b(x) = 6x^4 - 2x^3 + x, \qquad c(x) = 3x^3 - 7.$$

Since the degree of the result will be 7, we need $N = 8$. Then $\omega = 8$ is a primitive 8th root of unity. First, $\mathrm{FFT}(5, 8, 8)$ is simply $(5, 5, 5, 5, 5, 5, 5, 5)$ since evaluating a constant at any point will give the constant itself. For the remaining 3 polynomials, we have

$$\begin{aligned} A &= \mathrm{FFT}(a(x), 8, 8) = (-7, -8, 5, -2, -8, -7, -3, 4), \\ B &= \mathrm{FFT}(b(x), 8, 8) = (5, -2, -6, -3, 7, 7, 1, 8), \\ C &= \mathrm{FFT}(c(x), 8, 8) = (-4, -1, 5, 0, 7, 4, -2, 3). \end{aligned}$$

Then we compute $(A_i B_i - {C_i}^2 + 5 \mod 17)$ for $i = 0, 1, 2, \ldots, 7$. This gives the DFT of the final result:

$$(5, 3, 1, -6, 2, 8, -2, -6).$$

Now we compute the IDFT. As $\omega^{-1} = -2$, we perform

$$\mathrm{FFT}(\ (5 + 3x + x^2 - 6x^3 + 2x^4 + 8x^5 - 2x^6 - 6x^7)\ , 8, -2)$$

and get

$$(5, 8, -2, -3, 7, 5, 1, 2).$$

Thus, the desired polynomial result is

$$8^{-1}(5 + 8x - 2x^2 - 3x^3 + 7x^4 + 5x^5 + x^6 + 2x^7) = 7 + x + 4x^2 + 6x^3x^4 + 7x^5 - 2x^6 - 4x^7.$$

Before exhibiting an implementation of the FFT algorithm, we want to discuss how to determine a primitive n-th root of unity in a given field F. When $F = \mathbb{C}$, this is easy. A primitive n-th root of unity is simply $\omega = e^{2\pi i/n}$. However, when F is a finite field, say $F = \mathbb{Z}_p$ for some prime p, the situation is more complicated to deal with.

From finite field theory, we know that the multiplicative group of $\mathbb{Z}_p$ is actually a cyclic group. In other words, $(\mathbb{Z}_p, \times)$ is generated by a single element. Suppose α is such a generator. Since $\mathbb{Z}_p$ has exactly $p - 1$ distinct elements, we must have

$$\mathbb{Z}_p = \{1, \alpha, \alpha^2, \ldots, \alpha^{p-2}\} \quad \text{and} \quad \alpha^{p-1} = 1.$$

Now for any integer n that divides $p - 1$, $\alpha^{(p-1)/n}$ is an element in $\mathbb{Z}_p$. Since

$$(\alpha^{(p-1)/n})^n = \alpha^{p-1} = 1,$$

$\alpha^{(p-1)/n}$ is an n-th root of unity. In addition, for $0 < k < n$, we have $0 < (p-1)k/n < p - 1$. As a result, we get

$$\alpha^{(p-1)k/n} \neq 1,$$

which implies that $\alpha^{(p-1)/n}$ is also primitive. Hence $\omega = \alpha^{(p-1)/n}$ is a primitive n-th root of unity in $\mathbb{Z}_p$ whenever n divides $p-1$.

On the other hand, for an integer m which does not divide $p-1$, if there exists a primitive m-th root of unity, say δ, then $\{1, \delta, \delta^2, \ldots, \delta^{m-1}\}$ forms a cyclic subgroup of $(\mathbb{Z}_p, \times)$. By Lagrange's theorem from group theory (Gallian [3], 141-142), the order of this subgroup should divide the order of $\mathbb{Z}_p$. This implies m divides $p-1$, which results in a contradiction. So a primitive m-th root of unity cannot exist if m does not divide $p-1$.

In our FFT algorithm, we always need a primitive 2^k-th root of unity for some integer k. Hence we want a finite field $\mathbb{Z}_p$ where the prime p has the form

$$p = 2^r t + 1$$

for some $r \geqslant k$ and some odd number t. If such a finite field is given along with its generator α, then we may determine a primitive 2^k-th root of unity by computing

$$\omega = \alpha^{(p-1)/N}$$

with $N = 2^k$. In practice, we prefer a prime $p = 2^r t + 1$ with r as large as possible so as to deal with those polynomials of a huge degree. For example, $p = 2^{27} \cdot 15 + 1 = 2013265921$ is the largest 31-bit prime of such form.

Now we present the implementations of the fast Fourier transform algorithm and the associated multiplication algorithm.

The functions `add(a,b,p)`, `sub(a,b,p)` and `neg(a,b,p)` compute additions, subtractions and negations in $\mathbb{Z}_p$, respectively. They efficiently transform a negative number into the positive representation modulo p without an **if** statement.

```
inline int add( int a, int b, int p ) { int t; t = (a-p)+b; t += (t>>31) & p; return t; }
inline int sub( int a, int b, int p ) { int t; t = (a-b); t += (t>>31) & p; return t; }
inline int neg( int a, int p ) { int t; t = -a; t += (t>>31) & p; return t; }
```

The function `powmod(a,n,p)` computes $a^n \bmod p$ in $\mathbb{Z}_p$, for $0 \leqslant a < p < 2^{31}$.

```
int powmod( int a, int n, int p )
{   LONG r, s;
    for( r=1, s=a; n>0; n/=2 ) { if( n&1) r = (r*s)% p; s = (s*s)% p; }
    return r;
}
```

The function `polfill(A,da,n,A,B)` fills the array B, a copy of A, with dummy zero coefficients so that the length of B is extended to n. A is a polynomial of degree da in array representation.

```
void polfill( int *A, int da, int n, int *B )
{   int i;
    for( i=0; i<=da; i++ ) B[i] = A[i];
    for( ; i<n; i++ ) B[i] = 0;
    return;
}
```

The function `FFT(n,w,W,A,p,T)` performs the fast Fourier transform algorithm to compute the DFT of A in ${\mathbb{Z}_p}^n$, where A is a polynomial in array representation. We need n to be a power of 2 and w to be a primitive n-th root of unity. The array W stores the first $n/2$ Fourier points. T is a temporary array. We alternately use A and T to store intermediate results from the recursive calls in order to save space memory. The final result is put in A.

```
void FFT( int n, int w, int *W, int *A, int p, int *T )
{  int i, w1, n2;
   int  s, t, x;
   if ( n == 1 ) { return; }
   n2 = n/2;
   for( i=0; i<n2; i++ ) { T[i] = A[2*i]; T[n2+i] = A[2*i+1]; }
   FFT( n2, 2*w, W, T, p, A );
   FFT( n2, 2*w, W, T+n2, p, A+n2 );
   for( w1=0, i=0; i<n2; i++, w1+=w ) {
      t = T[n2+i];
      t = ((LONG) W[w1])*t % p;
      x = T[i];
      A[i] = add(x,t,p);
      A[n2+i] = sub(x,t,p); }
   return;
}
```

The function `FFTmul(A,B,T,da,db,alpha,W,p)` utilizes `FFT()` to compute a polynomial multiplication $T = A \cdot B$ in $\mathbb{Z}_p[x]$ where $\deg(A) = da$ and $\deg(B) = db$. The input integer *alpha* should be the generator of the field $\mathbb{Z}_p$. In consideration of space efficiency, we also use T to store the intermediate results of the `FFT()` calls. We compute $\{\omega, \omega^2, \ldots, \omega^{n/2-1}\}$ and store them in the array W.

```
int FFTmul( int *A, int *B, int *T, int da, int db, int alpha, int *W, int p )
{   int i, n, dc;
    dc = da+db;
    for( n=1; n<=dc; n*=2 );
    LONG w = powmod( alpha, (p-1)/n, p );
    W[0] = 1;
    for( i=1; i<n/2; i++ ) W[i] = (w*W[i-1])% p;
    polfill( A, da, n, T );
    FFT( n, 1, W, T, p, T+n );
    polfill( B, db, n, T+n );
    FFT( n, 1, W, T+n, p, T+2*n );
    for( i=0; i<n; i++ )  T[i] = ((LONG) T[i])*((LONG) T[i+n]) % p;
    for( i=1; i<n/4; i++ ) { int t = W[i]; W[i] = p-W[n/2-i]; W[n/2-i] = p-t; }
    W[n/4] = p-W[n/4];
    FFT( n, 1, W, T, p, T+n );
    LONG v = modinv(n, p);
    for( i=0; i<n; i++ ) T[i] = (v*T[i]) % p;
    return n;
}
```

5 Two Fast Power Series Inversion Algorithms

Now we are able to update our power series inversion algorithm by employing fast Fourier transforms as the multiplication method. Recall that we require two multiplications $y \cdot b$ and $y \cdot f$. In order to show the mechanism as clearly as possible, we give explicit steps for performing these two FFT multiplications.

Algorithm Fast Newton Inversion Algorithm

Input: A polynomial $a(x)$ with $a_0 \neq 0$ and a positive integer n.
Output: An order n approximation of $a(x)^{-1}$.

FNI$(a(x), n)$:

if $n = 1$ **then return** $y \leftarrow 1/a_0$

$m \leftarrow \lceil n/2 \rceil$

$y \leftarrow \text{FNI}((a(x), m)$

$b \leftarrow a(x) \mod x^n$

$M \leftarrow$ the first power 2 greater than $m + n - 2$

$\omega \leftarrow$ a primitive M-th root of unity

$W = (W_0, W_1, W_2, \ldots, W_{M/2-1}) \leftarrow (1, \omega, \omega^2, \ldots, \omega^{M/2-1})$

$(B_0, B_1, \ldots, B_{M-1}) \leftarrow \text{FFT}(b, M, W)$

$(Y_0, Y_1, \ldots, Y_{M-1}) \leftarrow \text{FFT}(y, M, W)$

for i from 0 to $M - 1$ **do** $G_i = B_i \cdot Y_i$

$V = (V_0, V_1, V_2, \ldots, V_{M/2-1}) \leftarrow (1, \omega^{-1}, \omega^{-2}, \ldots, \omega^{-(M/2-1)})$

$(g_0, g_1, \ldots, g_{M-1}) \leftarrow M^{-1} \cdot \text{FFT}(\sum_{i=0}^{M-1} G_i x^i, M, V)$

$\Big\}$ $\sum_{i=0}^{m+n-2} g_i x^i \leftarrow y \cdot b$

$f \leftarrow \sum_{i=0}^{n-m-1} (-g_{i+m}) x^i$

$N \leftarrow$ the first power 2 greater than $n - 2$ ▷ since $\deg(f) + \deg(y) \leqslant n - 2$

$j \leftarrow M/N$ ▷ j is either 1 or 2

$(F_0, F_1, \ldots, F_{N-1}) \leftarrow \text{FFT}(f, N, (W_0, W_j, W_{2j}, \ldots, W_{M/2-j}))$

for i from 0 to $N - 1$ **do** $H_i = F_i \cdot Y_{ij}$

$(h_0, \ldots, h_{N-1}) \leftarrow N^{-1} \cdot \text{FFT}(\sum_{i=0}^{N-1} H_i x^i, N, (V_0, V_j, V_{2j}, \ldots, V_{M/2-j}))$

$\Big\}$ $\sum_{i=0}^{n-2} h_i x^i \leftarrow y \cdot f$

$y \leftarrow y + \sum_{i=0}^{n-m-1} h_i x^{i+m}$

return y

Suppose we want to find an order n approximation of $a(x)^{-1}$ by FNI$(a(x), n)$ and N is a power of 2 such that $N/2 < n \leqslant N$. The second multiplication $y \cdot f$ can always be performed by FFTs of size N, whereas there are two possibilities for the first multiplication $y \cdot b$.

If we are lucky, say

$$n \leqslant n + \lceil n/2 \rceil - 1 \leqslant N,$$

we need three FFTs of size N to compute $y \cdot b$: one for the DFT of b, one for the DFT of y and the other for the IDFT of $y \cdot b$. Then computing $y \cdot f$ requires another two FFTs of size N: one for the DFT of f and the other for the IDFT of $y \cdot f$. Therefore, we need a total of 5 FFTs of size N. The multiplications' cost in this case is

$$5F(N) + O(N).$$

Otherwise, when

$$n \leqslant N < n + \lceil n/2 \rceil - 1,$$

the required size of FFTs for computing $y \cdot b$ increases to $2N$. Accordingly, the first multiplication requires three FFTs of size $2N$. For the second one, the DFT of f still costs an FFT of size N and IDFT of $y \cdot f$ costs another FFT of size N. Notice that it is unnecessary to compute $\mathrm{FFT}(y, N, (W_0, W_2, W_4, \ldots, W_{N-2}))$ since we have obtained $(Y_0, Y_1, \ldots, Y_{2N-1}) = \mathrm{FFT}(y, 2N, W)$. Hence the evaluations of y at N Fourier points are just

$$(y(1), y(\omega), \ldots, y(\omega^{N-1})) = (Y_0, Y_2, \ldots, Y_{2N-2}).$$

Consequently, we need 3 FFTs of size $2N$ and 2FFTs of size N in total. Correspondingly, the cost rises to

$$3F(2N) + 2F(N) + O(N).$$

Since $F(N) = cN \log N$ for some constant c, we have

$$F(2N) = 2cN \log 2N = 2cN(\log N + \log 2) = 2cN \log N + 2cN.$$

This means $F(2N) = 2F(N) + O(N)$ and thus the multiplication's cost in this case is

$$8F(N) + O(N).$$

We see that the original input n is critical to the efficiency of this algorithm. For convenience, we define a subset of $\mathbb{Z}^+$ by

$$L = \{n \in \mathbb{Z}^+ |\ 2^k < n \leqslant n + \lceil n/2 \rceil - 1 \leqslant 2^{k+1} \quad \text{for every } k \in \mathbb{N}\}. \tag{7}$$

Then, whether n belongs to L decides the practical cost of $\mathrm{FNI}(a(x), n)$. We claim this set membership is transitive, that is, $n \in L$ implies $\lceil n/2 \rceil \in L$. First, we deal with the base case. When $n = 2, \lceil n/2 \rceil = 1$. Clearly, $1 < n + \lceil n/2 \rceil - 1 = 2$ implies $2 \in L$. The following theorem guarantees the transitivity for $n \geqslant 3$.

Theorem 5.1. Let L be the set defined in (7). For any integer $n \geqslant 3$, if $n \in L$, then $\lceil n/2 \rceil \in L$.

Proof. When $n = 3, \lceil n/2 \rceil = 2$. We see that $2 < n + \lceil n/2 \rceil - 1 = 4$. So $3 \in L$. As $2 \in L$ is shown above, the statement is true for $n = 3$.

Now we turn to the general cases of $n \geqslant 4$. Let $m = \lceil n/2 \rceil$ and $\ell = \lceil m/2 \rceil$. Suppose $n \in L$, by definition, this means

$$n + m - 1 \leqslant 2^{k+1}$$

where $2^k < n \leqslant 2^{k+1}$. In order to prove $m \in L$, it is sufficient to show

$$2^{k-1} < m \leqslant 2^k \tag{8}$$

and

$$m + \ell - 1 \leqslant 2^k. \tag{9}$$

When n is even, we simply get $2^{k-1} < n/2 = m \leqslant 2^k$. When n is odd, we have $2^{k-1} < n < n + 1 \leqslant 2^k$, which implies $2^{k-1} < (n + 1)/2 = m \leqslant 2^k$. Hence inequation (8) holds.

Then we consider 4 cases of n: $4t, 4t + 1, 4t + 2$ and $4t + 3$ where $t \in \mathbb{N}$.

1. When $n = 4t$, we have $m = 2t$ and $\ell = t$. Then $n + m - 1 \leqslant 2^{k+1}$ means $6t - 1 \leqslant 2^{k+1}$. Since $6t - 1$ is odd, we get $6t - 1 < 6t \leqslant 2^{k+1}$. It follows that

$$m + \ell - 1 = 3t - 1 < 3t \leqslant 2^k.$$

2. When $n = 4t + 1$, we have $m = 2t + 1$ and $\ell = t + 1$. Then $n + m - 1 \leqslant 2^{k+1}$ means $6t + 1 \leqslant 2^{k+1}$. Again, since $6t + 1$ is odd, we have $6t + 1 < 6t + 2 \leqslant 2^{k+1}$, which gives

$$m + \ell - 1 = 3t + 1 \leqslant 2^k.$$

3. When $n = 4t + 2$, we have $m = 2t + 1$ and $\ell = t + 1$. Then $n + m - 1 \leqslant 2^{k+1}$ means $6t + 2 \leqslant 2^{k+1}$. Hence

$$m + \ell - 1 = 3t + 1 \leqslant 2^k.$$

4. When $n = 4t + 3$, we have $m = 2t + 2$ and $\ell = t + 1$. Then $n + m - 1 \leqslant 2^{k+1}$ means $6t + 4 \leqslant 2^{k+1}$. So we have

$$m + \ell - 1 = 3t + 2 \leqslant 2^k.$$

Thus, inequation (9) holds for all $n \geqslant 4$. This completes the proof.

□

By Theorem 5.1, if the original input n of $\mathrm{FNI}(a(x), n)$ belongs to L, we can deduce that all the integers

$$m_1, m_2, m_3, \ldots, 2$$

belong to L where $m_1 = \lceil n/2 \rceil$ and $m_{i+1} = \lceil m_i/2 \rceil$. Furthermore, if $2^{k-1} < n \leqslant 2^k$ for some $k \geqslant 1$, then we have

$$2^{k-1-i} < m_i \leqslant 2^{k-i}$$

for $1 \leqslant i \leqslant k - 1$. Then a recursive call $\mathrm{FNI}(a(x), m_i)$ requires $5F(2^{k-i}) + O(2^{k-i})$ cost to perform the two polynomial multiplications. Therefore, the total cost of $\mathrm{FNI}(a(x), n)$ when $n \in L$ is

$$I_f(n) = I_f(m_1) + 5F(2^k) + c2^k$$

$$
\begin{aligned}
&= I_f(m_2) + 5F(2^{k-1}) + c2^{k-1} + 5F(2^k) + c2^k \\
&\quad \vdots \\
&= I_f(1) + 5(F(2^k) + F(2^{k-1}) + \cdots + F(2)) + c(2^k + 2^{k-1} + \cdots + 2) \\
&= 1 + 5(F(2^k) + F(2^{k-1}) + \cdots + F(2)) + c(2^{k+1} - 2) \\
&< 5(F(2^k) + 2^{-1}F(2^k) + \cdots + 2^{-k+1}F(2^k)) + 2c2^k - 2c + 1 \\
&= 5F(2^k) \sum_{i=0}^{k-1} 2^{-i} + O(2^k) \\
&< 5F(2^k) \sum_{i=0}^{\infty} 2^{-i} + O(2^k) \\
&= 10F(2^k) + O(2^k).
\end{aligned}
$$

Nevertheless, when $n \notin L$, $\lceil n/2 \rceil \notin L$ is not necessarily true. For instance, $12 \notin L$ while $6 \in L$. Hence for the sequence of integers $m_1, m_2, m_3, \ldots, 2$ where $m_1 = \lceil n/2 \rceil$ and $m_{i+1} = \lceil m_i/2 \rceil$, some m_i could belong to L even if $n \notin L$. Once such an m_i exists, all the succeeding integers $m_{i+1}, m_{i+2}, \ldots, 2$ belong to L. This leads to the uncertainty of the total cost. For the sake of convenience, we need to conclude that such an m_i is rare.

Theorem 5.2. For every $k \in \mathbb{N}$, there exists at most one integer $n \in (2^k, 2^{k+1}]$, such that $n \notin L$ but $\lceil n/2 \rceil \in L$.

Proof. Let $m = \lceil n/2 \rceil$ and $\ell = \lceil m/2 \rceil$. If $n \notin L$, then we have

$$n + m - 1 > 2^{k+1}$$

where $2^k < n \leqslant 2^{k+1}$. As we shown in Theorem 5.1, m must satisfy

$$2^{k-1} < m \leqslant 2^k.$$

Then a sufficient condition for $m \notin L$ is

$$m + \ell - 1 > 2^k.$$

Again, we analyze 4 cases of n: $4t, 4t+1, 4t+2$ and $4t+3$ where $t \in \mathbb{N}$.

1. When $n = 4t + 1$, we have $m = 2t + 1$ and $\ell = t + 1$. Then we have

   $$6t + 2 > 6t + 1 = n + m - 1 > 2^{k+1},$$

   which gives

   $$m + \ell - 1 = 3t + 1 > 2^k.$$

   Hence $m \notin L$.

2. When $n = 4t + 2$, we have $m = 2t + 1$ and $\ell = t + 1$. Then $n + m - 1 > 2^{k+1}$ means $6t + 2 > 2^{k+1}$. Again, we get

$$m + \ell - 1 = 3t + 1 > 2^k.$$

So $m \notin L$.

3. When $n = 4t + 3$, we have $m = 2t + 2$ and $\ell = t + 1$. Then $n + m - 1 > 2^{k+1}$ means $6t + 4 > 2^{k+1}$. This leads to

$$m + \ell - 1 = 3t + 2 > 2^k$$

and thus $m \notin L$.

4. When $n = 4t$, we have $m = 2t$ and $\ell = t$. Then $n + m - 1 > 2^{k+1}$ means $6t - 1 > 2^{k+1}$. Since $6t - 1$ and 2^{k+1} are integers, we have $6t - 1 > 6t - 2 \geqslant 2^{k+1}$. It follows that

$$m + \ell - 1 = 3t - 1 \geqslant 2^k.$$

If $3t - 1 > 2^k$, we obtain $m \notin L$. Otherwise, if $3t - 1 = 2^k$, then $m \in L$ only when $3 \mid (2^k + 1)$. And

$$n = 4t = \frac{4(2^k + 1)}{3}$$

is the only n for this case.

□

According to this rarity, we may assume that if the initial input n of $\mathrm{FNI}(a(x), n)$ is not in L, then the integers $m_1, m_2, m_3, \ldots$ are not in L except those small ones such as $m_i = 2$. In this case, a recursive calls $\mathrm{FNI}(a(x), m_i)$ requires $8F(2^{k-i}) + O(2^{k-i})$ cost to perform the two polynomial multiplications. The total cost becomes

$$\begin{aligned} I_f(n) &= I_f(m_1) + 8F(2^k) + c2^k \\ &\vdots \\ &< 16F(2^k) + O(2^k). \end{aligned}$$

Now we consider the average cost of $\mathrm{FNI}(a(x), n)$. For $2^{k-1} < n \leqslant 2^k$, since

$$n + \lceil n/2 \rceil \approx \frac{3}{2}n,$$

if $n \in L$ then we approximately have

$$\frac{3}{2}n \leqslant 2^k.$$

Hence we obtain

$$2^{k-1} < n \leqslant \frac{4}{3}2^{k-1},$$

which means about $1/3$ of the integers in the interval $(2^{k-1}, 2^k]$ belong to L. If an n just exceeds 2^{k-1} and is far from 2^k, then $\mathrm{FNI}(a(x), n)$ will cost $10F(2^k) + O(2^k)$. Otherwise, its cost will rise to $16F(2^k)+O(2^k)$. Suppose we randomly pick an integer n to run $\mathrm{FNI}(a(x), n)$. After an adequate number of times, the total cost of this algorithm will be

$$\frac{1}{3}10F(2^k) + \frac{2}{3}16F(2^k) + O(2^k) = 14F(2^k) + O(2^k)$$

on average. If N denotes 2^k, then the average complexity of $\mathrm{FNI}(a(x), n)$ is

$$14F(N) + O(N).$$

In fact, there is an alternative way to compute $y(1-y\cdot b)$ by using fast Fourier transforms. Since the DFT is a homomorphism, we can do calculations directly after the transforms of y and b. Then we invert the result by the IDFT. As $\deg(y) \leqslant \lceil n/2 \rceil - 1$ and $\deg(b) \leqslant n-1$, the total degree of $y(1 - y \cdot b)$ must be at most

$$2\lceil n/2 \rceil + n - 3 \leqslant 2\frac{n+1}{2} + n - 3 = 2n - 2.$$

Again, say N is a power of 2 such that $N/2 < n \leqslant N$. Then we have

$$N - 2 < 2n - 2 \leqslant 2N - 2,$$

which means $2N$ is the first power of 2 greater than $2n - 2$. Therefore, it takes 3 FFT of size $2N$ to compute $y(1 - y \cdot b)$. Two for the DFTs of y, b and the other for the IDFT.

Algorithm Alternative Fast Newton Inversion Algorithm

Input: A polynomial $a(x)$ with $a_0 \neq 0$ and a positive integer n.
Output: An order n approximation of $a(x)^{-1}$.

ANI$(a(x), n)$:
if $n = 1$ **then return** $y \leftarrow 1/a_0$
$m \leftarrow \lceil n/2 \rceil$
$y \leftarrow \mathrm{ANI}((a(x), m)$
$b \leftarrow a(x) \mod x^n$
$M \leftarrow$ the first power 2 greater than $2n - 2$
$\omega \leftarrow$ a primitive M-th root of unity
$W \leftarrow (1, \omega, \omega^2, \ldots, \omega^{M/2-1})$
$(B_0, B_1, \ldots, B_{M-1}) \leftarrow \mathrm{FFT}(b, M, W)$
$(Y_0, Y_1, \ldots, Y_{M-1}) \leftarrow \mathrm{FFT}(y, M, W)$
for i from 0 to $M - 1$ **do** $R_i = Y_i(1 - B_i \cdot Y_i)$ ▷ the DFT of $y(1 - y \cdot b)$
$V \leftarrow (1, \omega^{-1}, \omega^{-2}, \ldots, \omega^{-(M/2-1)})$
$(h_0, h_1, \ldots, h_{M-1}) \leftarrow M^{-1} \cdot \mathrm{FFT}(\sum_{i=0}^{M-1} R_i x^i, M, V)$
$y \leftarrow y + \sum_{i=m}^{n-1} h_i x^i$ ▷ $h_0 = h_1 = \cdots = h_{m-1} = 0$
return y

The complexity of this alternative algorithm is clearer. We have the recurrence

$$I_a(n) = I_a(\lceil n/2 \rceil) + 3F(2N) + c(2N).$$

Solving it in the same manner as before, we obtain

$$\begin{aligned} I_a(n) &< 6F(2N) + O(2N) \\ &= 12F(N) + O(N). \end{aligned}$$

Compared with FNI$(a(x), n)$, ANI$(a(x), n)$ has an obvious advantage when $n \notin L$. Although it is less efficient when $n \in L$, on average it is a better choice than FNI$(a(x), n)$. The implementations of these two algorithms are given below.

The function `NewtonFast(a,n,T,y,p,alpha,W)` performs FNI(a, n) to compute $y = a^{-1} \mod x^n$ for a given $a \in \mathbb{Z}_p[x]$. The input array T is used as temporary storage for intermediate calculations. Its length should be at least $3M$ where M is the first power of 2 greater than $n + \lceil n/2 \rceil - 2$.

```
int NewtonFast( int *a, int n , int *T, int *y,  int p, int alpha, int *W )
{   if( n==1 ) { y[0] = modinv(a[0],p); return n; }
    int m = (n+1)/2;
    int N = NewtonFast( a, m, T, y, p, alpha, W);
    while( N<=m+n-2 )  N *= 2 ;
```

```
    LONG w = powmod(alpha, (p-1)/N, p);
    W[0] = 1; int i;
    for( i=1; i<N/2; i++ ) W[i] = (w * W[i-1]) % p;
    polfill( a, n-1, N, T );     FFT( N, 1, W, T, p, T+N );
    polfill( y, m-1, N, T+2*N ); FFT( N, 1, W, T+2*N, p, T+N );
    for( i=0; i<N; i++)  T[i] = ((LONG) T[i]) * ((LONG) T[i+2*N]) % p;
    for( i=1; i<N/4; i++ ) { int t = W[i]; W[i] = p-W[N/2-i]; W[N/2-i] = p-t; }
    W[N/4] = p-W[N/4];
    FFT(N, 1, W, T, p, T+N);
    LONG v = modinv(N, p);
    for( i=0; i<=n-m-1; i++ ) T[i] = neg(v*T[i+m] % p, p);
    for( i=1; i<N/4; i++ ) { int t = W[i]; W[i] = p-W[N/2-i]; W[N/2-i] = p-t; }
    W[N/4] = p-W[N/4];
    int j = 1;
    if( N/2>n-2 ) j = 2;
    N = N/j;
    for( i=1; i<N/2; i++ ) W[i] = W[i*j];
    polfill( T, n-m-1, N, T+N );     FFT( N, 1, W, T+N, p, T );
    for( i=0; i<N; i++ )  T[i] = ((LONG) T[i+N]) * ((LONG) T[i*j+2*N*j]) % p;
    for( i=1; i<N/4; i++ ) { int t = W[i]; W[i] = p-W[N/2-i]; W[N/2-i] = p-t; }
    W[N/4] = p-W[N/4];
    FFT( N, 1, W, T, p, T+N );
    for( i=0; i<=n-m-1; i++ ) y[i+m] = (j*v*T[i]) % p;
    return N*j;
}
```

The function `NewtonAlt(a,n,T,y,p,alpha,W)` performs $\mathrm{ANI}(a, n)$ to compute $y = a^{-1} \bmod x^n$ for a given $a \in \mathbb{Z}_p[x]$. The input array T is used as temporary storage for intermediate calculations. Its length should be at least $3M$ where M is the first power of 2 greater than $2n - 2$.

```
int NewtonAlt( int *a, int n , int *T, int *y,  int p, int alpha, int *W )
{   if( n==1 ) { y[0] = modinv(a[0],p); return n; }
    int i;
    int m = (n+1)/2;
    int N = NewtonAlt( a, m, T, y, p, alpha, W);
    while( N<=2*n-2 ) N *= 2;
    LONG w = powmod(alpha, (p-1)/N, p);
    W[0] = 1;
    for( i=1; i<N/2; i++ ) W[i] = (w * W[i-1]) % p;
    polfill( a, n-1, N, T );     FFT( N, 1, W, T, p, T+N );
    polfill( y, m-1, N, T+2*N ); FFT( N, 1, W, T+2*N, p, T+N );
    for( i=0; i<N; i++)
    T[i] = (1+p-((LONG) T[i])*((LONG) T[i+2*N])%p)*(((LONG) T[i+2*N])) % p;
    LONG v = modinv(N, p);
    for( i=1; i<N/4; i++ ) { int t = W[i]; W[i] = p-W[N/2-i]; W[N/2-i] = p-t; }
    W[N/4] = p-W[N/4];
    FFT( N, 1, W, T, p, T+N );
    for( i=m; i<=n-1; i++ ) y[i] = (v*T[i])%p;
    return N;
}
```

6 Middle Product Optimization

In the previous section, we see that the algorithm FNI$(a(x), n)$ takes 5 FFTs of size N to compute $(y(1 - y \cdot b) \mod x^n)$ in the lucky case, when $n \in L$. In fact, such efficiency can always be achieved independently of whether n belongs to L, by utilizing the middle product optimization

Recall that y is an order $m = \lceil n/2 \rceil$ approximation of $a(x)^{-1}$ and $b = a(x) \mod x^n$. Since $y \cdot a(x) = 1 \mod x^m$, we have

$$y \cdot b = 1 + 0 \cdot x + 0 \cdot x^2 + 0 \cdot x^3 + \cdots + 0 \cdot x^{m-1} + g_m \cdot x^m + g_{m+1} \cdot x^{m+1} + \cdots + g_{n+m-2} \cdot x^{n+m-2}$$

for some g_i's in F. In order to obtain

$$\begin{aligned} 1 - y \cdot b \mod x^n &= -g_m \cdot x^m - g_{m+1} \cdot x^{m+1} - \cdots - g_{n-2} \cdot x^{n-2} - g_{n-1} \cdot x^{n-1} \\ &= -x^m(g_m + g_{m+1} \cdot x + \cdots + g_{n-2} \cdot x^{n-m-2} + g_{n-1} \cdot x^{n-m-1}), \end{aligned}$$

we need to determine

$$g_m + g_{m+1} \cdot x + \cdots + g_{n-2} \cdot x^{n-m-2} + g_{n-1} \cdot x^{n-m-1}.$$

Definition 6.1. Suppose F is a field and $a(x)$ is a power series in F[[x]] with $a_0 \neq 0$. For some integer $n \geqslant 2$, let $b = a(x) \bmod x^n$, $\quad y = a(x)^{-1} \bmod x^m$ where $m = \lceil n/2 \rceil$. If

$$y \cdot b = \sum_{i=0}^{n+m-2} g_i x^i,$$

then the polynomial

$$\sum_{i=0}^{n-m+1} g_{m+i} x^i$$

is called the middle product, denoted by MP.

In FNI$(a(x), n)$, it takes 3 FFTs of size $2N$ to get MP because $\deg(y \cdot b) \leqslant n + m - 2$ and $N/2 < n \leqslant N$. In reality, we may just perform 3 FFTs of size N to determine MP.

Theorem 6.1. Let $F, a(x), n, b, m$ and y be the same as defined above. Also, Let N be a power of 2 such that $N/2 < n \leqslant N$ and ω be a primitive N-th root of unity. If

$$\begin{aligned} &\mathrm{FFT}(b, N, \omega) = (B_0, B_1, \ldots, B_{N-1}), \\ &\mathrm{FFT}(y, N, \omega) = (Y_0, Y_1, \ldots, Y_{N-1}), \\ &D_i = B_i Y_i \quad \text{for } i = 0, \ldots, N-1 \quad \text{and} \\ &(d_0, d_1, \ldots, d_{N-1}) = N^{-1}\mathrm{FFT}(\sum_{i=0}^{N-1} D_i x^i, N, \omega^{-1}), \end{aligned}$$

then we have

$$\mathrm{MP} = \sum_{i=0}^{n-m+1} d_{m+i} x^i.$$

Proof. By definition, if $y \cdot b = \sum_{j=0}^{n+m-2} g_j x^j$, then MP $= \sum_{i=0}^{n-m+1} g_{m+i} x^i$. So we need to show $g_j = d_j$ for each $j \in \{m, m+1, \ldots, n-1\}$.
Notice that when $n+m-2 < N$, this is just a normal FFT multiplication procedure, and hence the claim must be true. Then we consider the case of $n+m-2 \geqslant N$.
The first two FFTs evaluate b and y at the Fourier points

$$\{1, \omega, \omega^2, \ldots, \omega^{N-1}\}.$$

This means

$$(B_0, B_1, \ldots, B_{N-1}) = (b(1), b(\omega), \ldots, b(\omega^{N-1})),$$
$$(Y_0, Y_1, \ldots, Y_{N-1}) = (y(1), y(\omega), \ldots, y(\omega^{N-1})).$$

Since these evaluations are homomorphic, we have

$$\begin{aligned} D_i &= B_i Y_i \\ &= b(\omega^i) y(\omega^i) \\ &= (b \cdot y)(\omega^i) \\ &= \sum_{j=0}^{n+m-2} g_j (\omega^i)^j \end{aligned}$$

for $0 \leqslant i \leqslant N-1$.
As $n+m-2 \geqslant N$, we write

$$\begin{aligned} D_i &= \sum_{j=0}^{N-1} g_j (\omega^i)^j + \sum_{j=N}^{n+m-2} g_j (\omega^i)^j. \\ &= \sum_{j=0}^{N-1} g_j (\omega^i)^j + \sum_{j=0}^{n+m-2-N} g_{j+N} (\omega^i)^{j+N} \end{aligned}$$

Due to $w^N = 1$, we get

$$\begin{aligned} D_i &= \sum_{j=0}^{N-1} g_j (\omega^i)^j + \sum_{j=0}^{n+m-2-N} g_{j+N} (\omega^i)^j (\omega^i)^N \\ &= \sum_{j=0}^{N-1} g_j (\omega^i)^j + \sum_{j=0}^{n+m-2-N} g_{j+N} (\omega^i)^j (\omega^N)^i \\ &= \sum_{j=0}^{N-1} g_j (\omega^i)^j + \sum_{j=0}^{n+m-2-N} g_{j+N} (\omega^i)^j \end{aligned}$$

Since $m = \lceil n/2 \rceil \leqslant (n+1)/2 \leqslant (N+1)/2 = N/2 + 1/2$ and m is an integer, we know that $m \leqslant N/2$. Hence

$$N \geqslant 2m$$

$$-N \leqslant -2m$$
$$n+m-2-N \leqslant n-m-2$$

It follows that

$$n+m-2-N < m.$$

So we obtain

$$\begin{aligned} D_i &= \left(\sum_{j=0}^{n+m-2-N} g_j(\omega^i)^j + \sum_{j=n+m-1-N}^{m-1} g_j(\omega^i)^j + \sum_{j=m}^{N-1} g_j(\omega^i)^j \right) + \sum_{j=0}^{n+m-2-N} g_{j+N}(\omega^i)^j \\ &= \sum_{j=0}^{n+m-2-N} (g_j + g_{j+N})(\omega^i)^j + \sum_{j=n+m-1-N}^{m-1} g_j(\omega^i)^j + \sum_{j=m}^{N-1} g_j(\omega^i)^j \end{aligned}$$

Then the IDFT $(d_0, d_1, \ldots, d_{N-1}) = N^{-1}\mathrm{FFT}(\sum_{i=0}^{N-1} D_i x^i, N, \omega^{-1})$ interpolates the evaluation points $\{D_0, D-1, \ldots, D_{N-1}\}$ and returns

$$d_j = \begin{cases} g_j + g_{j+N} & \text{for } 0 \leqslant j \leqslant n+m-2-N, \\ g_j & \text{for } n+m-1-N \leqslant j \leqslant m-1, \\ g_j & \text{for } m \leqslant j \leqslant N-1. \end{cases}$$

Therefore, $d_j = g_j$ for each $j \in \{m, m+1, \ldots, n-1\}$. This completes the proof. □

Algorithm Fast Newton Inversion by Middle Product

Input: A polynomial $a(x)$ with $a_0 \neq 0$ and a positive integer n.
Output: An order n approximation of $a(x)^{-1}$.

MPNI$(a(x), n)$:
if $n = 1$ **then return** $y \leftarrow 1/a_0$
$m \leftarrow \lceil n/2 \rceil$
$y \leftarrow \mathrm{MPNI}((a(x), m)$
$b \leftarrow a(x) \mod x^n$
$N \leftarrow$ the first power 2 greater than $n-1$
$\omega \leftarrow$ a primitive N-th root of unity
$W \leftarrow (1, \omega, \omega^2, \ldots, \omega^{N/2-1})$
$(B_0, B_1, \ldots, B_{N-1}) \leftarrow \mathrm{FFT}(b, N, W)$
$(Y_0, Y_1, \ldots, Y_{N-1}) \leftarrow \mathrm{FFT}(y, N, W)$
for i from 0 to $N-1$ **do** $D_i = B_i \cdot Y_i$
$V \leftarrow (1, \omega^{-1}, \omega^{-2}, \ldots, \omega^{-(N/2-1)})$
$(d_0, d_1, \ldots, d_{N-1}) \leftarrow N^{-1} \cdot \mathrm{FFT}(\sum_{i=0}^{N-1} D_i x^i, N, V)$
$\mathrm{MP} \leftarrow \sum_{i=0}^{n-m-1} (d_{m+i}) x^i$
$(F_0, F_1, \ldots, F_{N-1}) \leftarrow \mathrm{FFT}(-\mathrm{MP}, N, W)$
for i from 0 to $N-1$ **do** $H_i = F_i \cdot Y_i$
$(h_0, h_1, \ldots, h_{N-1}) \leftarrow N^{-1} \cdot \mathrm{FFT}(\sum_{i=0}^{N-1} H_i x^i, N, V)$
$y \leftarrow y + \sum_{i=0}^{n-m-1} h_i x^{i+m}$
return y

The algorithm MPNI$(a(x), n)$ computes $(y(1 - y \cdot b) \mod x^n)$ by using 5 FFTs of size N where $N/2 < n \leqslant N$. Hence we obtain

$$I_m(n) = I_m(\lceil n/2 \rceil) + 5F(N) + O(N).$$

We have shown that this recurrence leads to

$$I_m(n) < 10F(N) + O(N).$$

In comparison with ANI$(a(x), n)$, the time efficiency is improved by 1/6, which is not so remarkable. If we take space efficiency into consideration, then MPNI$(a(x), n)$ shows an advantage. It requires half as much space as ANI$(a(x), n)$ does. The implementation of MPNI$(a(x), n)$ is presented below.

The function `NewtonMP(a,n,T,y,p,alpha,W)` performs MPNI(a, n) to compute $y = a^{-1} \mod x^n$ for a given $a \in \mathbb{Z}_p[x]$. The input array T is used as temporary storage for intermediate calculations. Its length should be at least $3N$ where N is the first power of 2 greater than $n-1$.

```
int NewtonMP( int *a, int n, int *T, int *y,int p, int alpha, int *W )
{   if( n==1 ) {y[0] = modinv(a[0], p); return n; }
    int m = (n+1)/2;
    int N = NewtonMP( a, m, T, y, p, alpha, W );
    N *= 2 ;
    LONG w = powmod(alpha, (p-1)/N, p);
    W[0] = 1;
    int i;
    for( i=1; i<N/2; i++ ) W[i] = (w * W[i-1]) % p;
    polfill( a, n-1, N, T );     FFT( N, 1, W, T, p, T+N );
    polfill( y, m-1, N, T+2*N );
    FFT( N, 1, W, T+2*N, p, T+N );
    for( i=0; i<N; i++)  T[i] = ((LONG) T[i]) * ((LONG) T[i+2*N]) % p;
    for( i=1; i<N/4; i++ ) { int t = W[i]; W[i] = p-W[N/2-i]; W[N/2-i] = p-t; }
    W[N/4]=p-W[N/4];
    FFT( N, 1, W, T, p, T+N );
    LONG v = modinv(N, p);
    for(i=0;i<=n-m-1;i++) T[i]= neg((v*T[i+m]) % p, p);
    for( i=1; i<N/4; i++ ) { int t = W[i]; W[i] = p-W[N/2-i]; W[N/2-i] = p-t; }
    W[N/4]=p-W[N/4];
    polfill( T, n-m-1, N, T+N );
    FFT( N, 1, W, T+N, p, T );
    for( i=0; i<N; i++)  T[i] = ((LONG) T[i+N]) * ((LONG) T[i+2*N]) % p;
    for( i=1; i<N/4; i++ ) { int t = W[i]; W[i] = p-W[N/2-i]; W[N/2-i] = p-t; }
    W[N/4]=p-W[N/4];
    FFT( N, 1, W, T, p, T+N );
    for( i=0; i<=n-m-1; i++) y[i+m] = (v*T[i]) % p;
    return N;
}
```

7 Timings

Now we test the time efficiency of our power series inversion algorithms. For different integers n, we generate a random polynomial $a(x)$ of degree $n-1$ in $\mathbb{Z}_p[x]$ where

$$p = 2013265921 = 2^{27} \cdot 3 \cdot 5 + 1.$$

We do timings for running the functions `NewtonSlow()`, `NewtonFast()`, `NewtonAlt()` and `NewtonMP()` to compute an order n approximation of $a(x)^{-1}$.

n	if $n \in F$	NewtonSlow	NewtonFast	NewtonAlt	NewtonMP
1024	No	12ms	13ms	10ms	7ms
2048	No	49ms	27ms	22ms	16ms
4100	Yes	188ms	67ms	96ms	71ms
6600	No	488ms	120ms	96ms	72ms
8500	Yes	812ms	153ms	204ms	152ms
10000	Yes	1192ms	163ms	209ms	154ms
12000	No	1604ms	258ms	205ms	153ms
16384	No	2993ms	259ms	205ms	154ms
18000	Yes	3626ms	328ms	437ms	329ms
20000	Yes	4583ms	338ms	452ms	344ms
32768	No	12361ms	577ms	454ms	340ms

We see that the time for `NewtonSlow()` is quadratic and grows much more rapidly than those of the three FFT-based algorithms. As we expected, when $n \in L$, `NewtonFast()` and `NewtonMP()` almost require the same running time and are both faster than `NewtonAlt()` . When $n \notin L$, `NewtonAlt()` is faster than `NewtonFast()` but still slower than `NewtonMP()`.

We construct a table of the time cost ratios to illustrate the efficiency.

n	if $n \in F$	NewtonFast : NewtonAlt	NewtonMP : NewtonAlt
1024	No	1.300	0.700
2048	No	1.227	0.727
4100	Yes	0.698	0.739
6600	No	1.250	0.750
8500	Yes	0.750	0.745
10000	Yes	0.780	0.737
12000	No	1.258	0.746
16384	No	1.263	0.751
18000	Yes	0.751	0.753
20000	Yes	0.748	0.761
32768	No	1.271	0.749

As we discussed in section 4, the complexity of `NewtonFast()` is

$$I_f(n) = 10F(N) + O(N) \text{ when } n \in L$$

and

$$I_f(n) = 16F(N) + O(N) \text{ when } n \notin L.$$

The complexity of `NewtonAlt()` is always

$$I_a(n) = 6F(2N) + O(N) > 12(N) + O(N)).$$

And the complexity of `NewtonMP()` is always

$$I_m(n) = 10F(N) + O(N).$$

Theoretically, we should have

$$I_f(n) : I_a(n) < 10 : 12 \approx 0.83 \quad \text{when } n \in L$$

and

$$I_f(n) : I_a(n) < 16 : 12 \approx 1.33 \quad \text{when } n \notin L.$$

For any n, we should have

$$I_m(n) : I_a(n) < 10 : 12 \approx 0.83.$$

Our timing results show that

$$\begin{aligned} &I_f(n) : I_a(n) \approx 0.75 \quad \text{when } n \in L, \\ &I_f(n) : I_a(n) \approx 1.25 \quad \text{when } n \notin L \text{ and} \\ &I_m(n) : I_a(n) \approx 0.75, \end{aligned}$$

which substantially matches the expectation.

8 Conclusion

The following table summarizes the complexity of the power series inversion algorithms in this paper. We assume that N is the first power of 2 greater than n. Also, $F(N) \in O(N \log N)$.

Algorithm	NewtonSlow	NewtonFast (average)	NewtonAlt	NewtonMP
Complexity	$2/3n^2 + O(n)$	$14F(N) + O(N)$	$12F(N) + O(N)$	$10F(N) + O(N)$
Space for T	n	$5N$	$6N$	$3N$
Space for W	0	$5/6N$	N	$N/2$

Since MPNI(a, n) is the best algorithm in terms of time complexity, we should adopt it as the power series inversion method in our polynomial division procedure. Once we get the inverse of the reciprocal polynomial of the divisor, we use FFTMul(a, b) to compute the quotient and remainder. Finally, we obtain a fast polynomial division algorithm whose complexity is in $O(N \log N)$.

Algorithm Fast Polynomial Division

Input: $a, b \in F[x]$ where F is a field and $b \neq 0$.

Output: $q, r \in F[x]$ satisfying $a = b \cdot q + r$ where $r = 0$ or $\deg(r) < \deg(b)$.

$k \leftarrow \deg(a) - \deg(b)$ ▷ $k = \deg(q)$

if $k < 0$ **then return** $q = 0, r = a$

if $\deg(b) = 0$ **then return** $q = a/b_0, r = 0$

$c \leftarrow \text{MPNI}(b^r, k)$

$d \leftarrow \text{FFTMul}(a^r, c) \mod x^{k+1}$

$\ell \leftarrow \deg(d)$

$q \leftarrow x^{k-\ell} d^r$

$r \leftarrow a - \text{FFTMul}(b, q)$

return q, r

References

[1] K.O. Geddes, S.R. Czapor and G. Labahn. (1992). *Algorithms for Computer Algebra.* Norwell, MA: Kluwer Academic Publishers.

[2] Guillaume Hanrot, Michel Querica and Paul Zimmermann. The Middle Product Algorithm, I. *Applicable Algebra in Engineering, Communication and Computing.* Volume 14, Issue 6, pp 415438, March 2004.

[3] Joseph A. Gallian. (2010). *Contemporary Abstract Algebra.* (7^{th} ed.). Belmont, CA: Brooks/Cole.

[4] Richard L. Burden and J. Douglas Faires. (2011). *Numerical Analysis.* (9^{th} ed.). Belmont, CA: Brooks/Cole.