NTRU Cryptosystems Technical Report

Report # 010, Version 1
Title: High-Speed Multiplication of (Truncated) Polynomials
Author: Joseph H. Silverman
Release Date: Tuesday, January 5, 1999

Abstract. Multiplication of two (truncated) polynomials of degree n takes on the order of n^2 operations. By splitting the polynomials into two pieces, this may be reduced to approximately $\frac{3}{4}n^2$ operations, and repeated recursive application of this procedure leads to even greater savings.

It is possible to reduce the number of operations needed to multiply two polynomials by successively splitting them in half. The basic idea is described, for example, in [1, section 3.1.2]. In this note we explain how this recursive splitting procedure works, describe some experimental results showing the possible savings, and give a pseudo-code implementation.

Let b and c be polynomials of degree $n-1$. The naive formula for the product $a = bc$ is

$$a_k = \sum_{i=\max(0,k-n+1)}^{\min(k,n-1)} b_i c_{k-i}, \qquad 0 \le k < 2n-1.$$

This involves approximately n^2 operations (where an operation consists of one multiplication and one addition).

The successive splitting idea begins by writing b and c as sums. Let $n_1 = \lfloor n/2 \rfloor$ and $n_2 = n - n_1$, and write

$$b = b_1 + b_2 X^{n_1} \qquad \text{and} \qquad c = c_1 + c_2 X^{n_1}$$

with

$$\deg(b_1) = \deg(c_1) = n_1 - 1 \qquad \text{and} \qquad \deg(b_2) = \deg(c_2) = n_2 - 1.$$

The product bc is then equal to

$$b * c = b_1 c_1 + (b_1 c_2 + b_2 c_1) X^{n_1} + b_2 c_2 X^{2n_1}.$$

This involves computing four products $b_1c_1, b_1c_2, b_2c_1, b_2c_2$ of polynomials of degree (approximately) $n/2$, so a total of $4 \cdot (n/2)^2 = n^2$ operations. Nothing has been gained.

However, the middle coefficient can be rewritten as

$$b_1 c_2 + b_2 c_1 = (b_1 + b_2)(c_1 + c_2) - b_1 c_1 - b_2 c_2.$$

Since the products b_1c_1 and b_2c_2 have already been computed, this reduces the number of products from four to three. So at the cost of a few extra additions, the computation now takes only $3 \cdot (n/2)^2 = \frac{3}{4}n^2$ operations.

To recapitulate, the product of $b = b_1 + b_2X^{n_1}$ and $c = c_1 + c_2X^{n_1}$ can be computed as follows:

- $a_1 = b_1c_1$
- $a_2 = b_2c_2$
- $a_3 = (b_1 + b_2)(c_1 + c_2)$
- $a = a_1 + (a_3 - a_1 - a_2)X^{n_1} + a_2X^{2n_1}$

If now this process is applied recursively r times, then the number of operations is reduced to approximately $\left(\frac{3}{4}\right)^r n^2$. Of course, there is additional overhead incurred when invoking a subroutine (recursive or not), and the recursive routine includes extra steps not required by the naive multiplication formula. In practice, one determines the optimal number of recursions experimentally.

The results in Table 1 were obtained on a 333 MHz Macintosh G3 using Metroworks CodeWarrior C compiler. There was no real effort made to write optimized code. The naive multiplication was performed within the main routine. The recursive multiplication was done by recursive calls to a subroutine until the degree of the subdivided polynomials was less than the cutoff value d. The polynomials b and c had degree $n - 1$. The coefficients of b were randomly chosen from the set $\{-1, 0, 1\}$, and the coefficients of c were randomly chosen between 0 and 255. The time listed is the average time (in milliseconds) for a single product, averaged over between 1000 and 10000 samples.

n	Naive (ms)	Recursive (ms)				
		$d = 8$	$d = 16$	$d = 32$	$d = 64$	$d = 128$
503	9.41	4.97	3.95	3.80	4.67	5.67
263	2.58	1.79	1.41	1.33	1.38	1.64
107	0.435	0.335	0.295	0.305	0.348	—

Table 1. Time to Perform Polynomial Multiplications

The table shows that significant time savings are available using recursive multiplication. For example, time was reduced by almost a factor of 2.5 for polynomials with $n = 503$ by taking a cutoff of $d = 32$.

The following pseudo-code description will assist in implementing the recursive multiplication routine.

```
PolyMult(a,b,c,n,N) {
[Description. Compute the product b*c and store the result in a. The polynomials
b and c have degrees n-1. The routine is called recursively until the polynomials
have degrees less than a preset CutOff parameter. If N > 0, then the product is
computed as a convolution, using the relation X^N = 1. ]
  /* if n is small, compute the product directly */
  if ( n < CutOff ) {
      for ( k=0; k<=2*n-2; k++ ) {
          a[k]=0;
          for ( i=max(0,k-n+1); i<=min(k,n-1); i++) {
              a[k] += b[i]*c[k-i];
          }
      }
  }
  /* otherwise n is large, compute the product recursively */
  else {
      n1 = n/2; /* n1 = 1 + degree of b1 and c1 */
      n2 = n - n1; /* n2 = 1 + degree of b2 and c2 */
      Write b as b = b1 + b2*X^n1;
      Write c as c = c1 + c2*X^n1;
      B = b1 + b2 /* note B has degree n2-1 */
      C = c1 + c2 /* note C has degree n2-1 */
      PolyMult(a1,b1,c1,n1,N); /* a1 = b1*c1 */
      PolyMult(a2,b2,c2,n2,N); /* a2 = b2*c2 */
      PolyMult(a3,B,C,n2,N); /* a3 = B*C = (b1+b2)*(c1+c2) */
      a = a1 + (a3-a1-a2)*X^n1 + a2*X^(2*n1);
  }
  /* the degree of a will be 2*n-2.  if this is larger than N and */
  /* N > 0, then use the relation X^N = 1.  (we assume n <= N.) */
  if ( 2*n-1 > N && N > 0 ) {
      for ( k=N; k<2*n-1; k++ )
          a[k-N] += a[k];
  }
  return 0;
}
```

Remark. If the polynomial `b` in the `PolyMult` routine tends to have small coefficients, then it is worthwhile changing the innermost assignment statement

```
a[k] += b[i]*c[k-i];
```

to take advantage of the likelihood that `b[i]` will be small. For example, one might replace the above assignment with the following:

```
switch (b[i]) {
    case 0 :  break;
    case 1 :  a[k] += c[k-i]; break;
    case -1 :  a[k] -= c[k-i]; break;
    default :  a[k] += b[i]*c[k-i]; break;
};
```

Remark. In order to avoid copying numbers in and out of memory more than necessary, one should perform polynomial splitting by using pointers. For example, let b be a polynomial of degree $n-1$ whose coefficients are given by the list

$$b = [\beta_0, \beta_1, \beta_2, \ldots, \beta_{n-1}],$$

and let $n_1 = \lfloor n/2 \rfloor$. In order to write b in the form $b = b_1 + b_2 X^{n_1}$, it suffices to let b_1 be a pointer to β_0 and let b_2 be a pointer to β_{n_1}. In other words, if b is a pointer to a list of length n, then $b_1 = b$ is a pointer to a list of length n_1, and $b_2 = b + n_1$ is a pointer to a list of length $n - n_1$.

References

[1] H. Cohen, *A course in computational algebraic number theory*, Graduate Texts in Math., vol. 138, Springer Verlag, Berlin, 1993

Comments and questions concerning this technical report should be addressed to

`techsupport@ntru.com`

Additional information concerning NTRU Cryptosystems and the NTRU Public Key Cryptosystem are available at

`www.ntru.com`
