Multiplying large numbers and the Schönhage-Strassen Algorithm

by Theo Kortekaas

Introduction

Making multiplications we do no longer out of our minds. It is often done for us. Consider the checkout in the supermarket, as we have to pay seven bottles of beer 79 cents each. If we have to multiply or calculate something ourselves then we use a calculator. For more elaborate calculations, we use a spreadsheet on a computer.
With computer programs like Excel or OpenOffice Calc we can multiply together numbers of about eight digits. The result is a number of about 15 digits, and that figure is exactly correct. This is sufficient for most applications. For larger numbers, a rounding is applied.

However, there are areas where one must be able to make multiplications with much larger numbers; and the result should be exactly correct. As with cryptographic applications, where large prime numbers are used; numbers of up to hundreds of decimal digits. Large numbers are also used in the field of mathematics and number theory, to calculate as many digits of π (pi) or *e* (the base of natural logarithms). There is also a race to find all but larger prime numbers. The largest prime number at this time (begin 2015) has more than seventeen million decimal digits (Mersenne prime). In all of these applications, there should be an ability to make calculations and multiplications by very large numbers. It is clear that we can only do this using a computer

Complexity

The larger the numbers, the more effort it is to multiply these numbers. But how much more? Scientists have established a measure for this effort: the Big-O notation [6]. This notation is called complexity and is the ratio of the amount of work to carry out for a formula or a particular algorithm with respect to the input to this formula or algorithm. The input is usually represented as n. In the formulas and algorithms that are used in large multiplications, n is usually expressed in terms of number of decimal digits. In computer applications the input size n is also commonly expressed in number of bits of the numbers, which are to be multiplied. An implicit assumption is that these numbers are about the same size.

So, the complexity of an addition of two numbers with n decimals each is $O(n)$. This means that if the numbers of a sum are twice as large (i.e., n is twice as large), that then also the amount of work in order to carry out the addition will be twice as large. Multiplication as we have learned in school (which we call here the classical method) has a complexity of $O(n^2)$. That means that if the numbers to multiply are two times as large, the amount of work is four times as large.

Multiplication with a computer program

For multiplying numbers of up to about eight decimal digits, for most computers standard programs are available such as OpenOffice Calc or Excel. For multiplying larger numbers,

special programs are needed, or special subroutines that can handle large numbers. These subroutines used to work in the traditional way (classical method). However, if the numbers are becoming very large, the time it takes to multiply these numbers with the classical method, even on fast computers, will still be unacceptably long. Therefore, Science (mathematicians and computer specialists) sought other methods to accelerate the calculation. This has led to special algorithms that can yield spectacular acceleration. That is what this document is about.

The algorithms discussed here are: the Karatsuba algorithm, the Toom-Cook algorithm and the Schönhage-Strassen algorithm (SSA). For the latter an implementation for a 32-bit Windows system is described. The three algorithms are compared with the classical method and with each other.

One of the newer developments is the Fürer algorithm. This algorithm is even faster than SSA, but only for astronomically large numbers. In practice, this algorithm is not used. Therefore, it is not included here.

Classical method

Let's look at the method we learned in school (what we called the classical method). Suppose we want to multiply two numbers with four decimal digits each (1234 x 5678). Normally, at the first multiplication (8 x 4 = 32) we would write down 2 and remember 3 etc. Now let's do something else: we write 32 in its entirety (we call this a '*core multiplication*') and provide more space between the columns. It goes like this:

```
             1  2  3  4
             5  6  7  8  x
             ----------
             8 16 24 32
          7 14 21 28
       6 12 18 24
    5 10 15 20
   --------------------
    5 16 34 60 61 52 32
```

The columns represents each an amount of ten to a power as follows
(remember that $10^0 = 1$) :

```
   32 x   10^0 =         32
   52 x   10^1 =        520
   61 x   10^2 =      6,100
   60 x   10^3 =     60,000
   34 x   10^4 =    340,000
   16 x   10^5 =  1,600,000
    5 x   10^6 =  5,000,000
                 ----------
 1234 x 5678 =  7,006,652
```

We see that there are sixteen *core multiplications* performed in this example. If we instead of two numbers with four decimal digits each, would have taken two numbers with with six decimal digits, then the number of *core multiplications* would have been 36. With two numbers of eight decimal digits each, the number of *core multiplications* would even have been 64. The amount of work in order to multiply two decimal numbers (expressed in number of *core-multiplications*), therefore, is by the classic method equal to the square of the number of decimal digits, of which the numbers consist. In short, the complexity is $O(n^2)$. Here we assume that the numbers are of approximately equal size.

Core multiplication

We use this term to indicate a multiplication on the lowest level. In the above example, the multiplication of 4 by 8 = 32 is a core multiplication because we have learned this in school with the multiplication tables, and we can not bring it down to an even simpler level. When we talk about a computer than we mean by a core multiplication: the multiplication of two integers of the word length that the computer uses. Word length is expressed in number of bits and is for the most common computers at present 32 bits or 64 bits. The multiplication of a 32-bit integer with a another 32-bit integer yields a result of up to 64 bits and can usually be done in one computer instruction.

Divide and Conquer (D&C)

One method to tackle complicated mathematical and technical issues is: to divide the problem into smaller and smaller pieces until a level is reached at which the sub-problem is clear and easy to solve. Thereafter, the partial solutions are combined (integrated) so that the final solution is found. In the literature such a method is called Divide and Conquer [12].This technique is also applicable to large multiplications. Let us take as illustration the example already worked out from the section on the classical method: 1234 x 5678.
We will now divide both numbers into two parts and name them as follows:

```
1  2  = a                  3  4  = b
5  6  = c                  7  8  = d
```

(We use hier the symbol * to denote multiply). The first number we can write as (a*10^2+b) and the second as (c*10^2+d). We can now apply some algebra on the multiplication:
(a*10^2+b) x (c*10^2+d) = ac*10^4 + (ad + bc)*10^2 + bd.
The result is four smaller multiplications and a number of additions:

```
a*c =   1  2      a*d =   1  2      b*c =   3  4      b*d =   3  4
        5  6 x            7  8 x            5  6 x            7  8 x
      -----             -----             -----             -----
        6 12              8 16             18 24             24 32
     5 10              7 14             15 20             21 28
    --------          --------          --------          --------
     5 16 12           7 22 16          15 38 24          21 52 32
```

Now we count everything together but we need to remember that a and c still have to be multiplied by 100. Thus, the result of ac must be magnified by a factor of 100 x 100 (10^4), and the results of ad and bc must be magnified by a factor of 100 (10^2).

```
                                                       32 x 10^0 =    32
                                     24 x 10^0 =    24    52 x 10^1 =   520
                  16 x 10^0 =  16    38 x 10^1 =   380    21 x 10^2 = 2,100
12 x 10^0 =  12   22 x 10^1 = 220    15 x 10^2 = 1,500               ------
16 x 10^1 = 160    7 x 10^2 = 700               ------                2,652
 5 x 10^2 = 500              ----         1,904 x 10^2 =            190,400
       ----              936 x 10^2  =                               93,600
      672 x 10^4 =                                                6,720,000
                                                                 ----------
                                                  Total =         7,006,652
```

The total is equal to the result of the classical method, as is the intention. However, also the number of core-multiplications is 16 and therefore equal to the number of the classical method, and the number of additions is larger. Do we gain something with this procedure? Later on we will see that there are methods to reduce the number of multiplications.

Recursion

We can imagine a computer program that gets two numbers as input; these numbers divides into two parts (*a* and *b*, *c* and *d*) and performs four multiplications with this parts (*ac*, *ad*, *bc*, and *bd*); the results of these multiplications combines in the way, as described in the here-above example, and the result presents as final output.
We can start with two numbers of 8 digits each. These numbers will be split into two four-digit numbers each so we get four four-digit numbers in total. To get the end result of 8 x 8 digits, we must perform four multiplications of 4 x 4 digits.
These four multiplications, we can also perform recursively by the same computer program. The result is that we have 4x4 = 16 multiplications of two by two digits. A level deeper we get 64 multiplications of one digit by one digit. in a number. This is the lowest level; a one digit figure can not be further divided. At this level we apply the multiplication tables as we have learned in school and we have given the name core multiplication.
A multiplication of 8 digits by 8 digits leads to 8x8 = 64 core multiplications. That is also consistent with the complexity of the classical multiplications as we have seen earlier, namely: $O(n^2)$.

Of course in a computer programs we will not use as the lowest level a multiplication of one decimal digit by one decimal digit, but the unit of a computer word by a computer word. Such multiplication is usually performed in one computer instruction.
For smaller numbers the classical method of multiplication still is the recommended method. The more advanced algorithms has a speed advantage for larger numbers. We will show that later on.

The algorithm of Karatsuba

Until 1960, everyone thought that the effort for multiplying two numbers increased with the square of the number of digits of the numbers to multiply. So if the numbers are twice as large, it costs four times as much work. Thus, the complexity is *$O(n^2)$*. This was in 1960 also claimed by Andrey Kolmogorov, a Russian mathematician, at a seminar on mathematical problems at the University of Moscow. [10]
However Anatolii Karatsuba, a then 23-year-old Russian student who attended the seminar, found within a week a faster algorithm, with which he undermined the claim of Kolmogorov. Karatsuba let him know his algorithm and Kolmogorov was "not amused". He told the next lesson of the finding of Karatsuba and canceled the rest of the seminar.
To illustrate the idea of Karatsuba we can best use the calculation of the section "Divide and Conquer". The formula, after we had split the first number into two parts: *a* and *b*, and the second number in *c* and *d,* was:

```
(a+b) x (c+d) = ac + ad + bc + bd = ac + (ad+bc) + bd.
Additional ac must be multiplied with 10^4 and
(ad+bc) with 10^2.
```

Karatsuba figured out the following:
When you add *a* and *b* together as well as *c* and *d* and you multiply the results of these sums you get the value (*ac*+*ad*+*bc*+*bd*) in one number; When you also multiply *a* with *c* (*ac*) and *b* with *d* (*bd*) then you have three numbers: (*ac*), (*bd*), and (*ac*+*ad*+*bc*+*bd*).
You can now reduce (*ac*+*ad*+*bc*+*bd*) with the value of (*ac*) and with the value of (*bd*) and the value of (*ad* +*bc*) is left in one number. Now you have all the ingredients to complete the foregoing formula.
Example: Take the numbers from the example of the Classical Method

```
1  2  = a                 3  4  = b
5  6  = c                 7  8  = d
```

We count a and b together and c and d:

```
a + b =  12 + 34 =  46
c + d =  56 + 78 = 134
```

We will now perform three multiplications;

```
a*c:      1  2     (a+b)*(c+d):   4  6      b*d:     3  4
          5  6 x               1  3  4 x             7  8 x
          ----                 -------              -----
          6 12                   16 24              24 32
       5 10                   12 18              21 28
                           4  6
       -------              -----------          --------
       5 16 12               4 18 34 24          21 52 32
Subtract(ac)                     5 16 12 -
Subtract(bd)                  21 52 32 -
                             ----------
                             2  8  4  0
```

As with normal subtraction we can now borrow from the column to the left. However, if we have more than 10 short in a column, we have to loan more than one from the column to the left. Now we count everything together but we need to remember that *a* and *c* have to be multiplied by 100. Thus, the result of *ac* must be multiplied by a factor of 100 x 100 (10^4), and the results of *ad* and *bc* must be multiplied by a factor of 100 (10^2).

```
     ac:                  (a+b)*(c+d):              bd:

                        0 x 10⁰ =     0       32 x 10⁰ =     32
                        4 x 10¹ =    40       52 x 10¹ =    520
12 x 10⁰ =    12        8 X 10² =   800       21 x 10² =  2,100
16 x 10¹ =   160        2 x 10³ = 2,000                   -----
 5 x 10² =   500                  -----                   2,652
          -----                   2,840 x 10² =         284,000
            672 x 10⁴ =                               6,720,000
                                                      ---------
Total:                                                7,006,652
```

We now have performed 12 core multiplications and some additions and subtractions. The multiplication of 1 x 46 in $(a + b) * (c + d)$ we do not count; as a result of the addition of two numbers of two digits there may be a third digit in the result but this only can be a one. 1 x 46 does not have to be treated as a multiplication but can be treated as an addition. Instead of the three times four multiplications we can perform three times the Karatsuba algorithm recursively. Then the total number of core multiplications is 3 x 3 = 9.
In principle, we can repeatedly perform the recursive Karatsuba algorithm until we reach the level of core multiplication (one decimal digit times one decimal digit, or one computer word times one computer word). With the Karatsuba algorithm each doubling of the number of digits requires a threefold number of multiplications and not four times as in the conventional method. This results in a complexity of $O(n^{\log 3})$, or about $O(n^{1.585})$ [10].
That is why the method of Karatsuba for larger numbers is much faster than the classical method. However, because there are quite a few additional operations (like additions and subtraction) required, the Karatsuba method for smaller numbers may be slower than the Classical method. This is also true for other algorithms (still to be discussed). What the turning points are will be discussed in next chapters.

Because the factors (numbers) to be multiplied with the Karatsuba algorithm should be halved each level of recursion, it is necessary that these factors exist of a number of digits that is a power of two. If the numbers do not have a power of two digits, they can be filled with zeroes until the next power of two is reached.

Toom-Cook Algorithm

This algorithm has been developed by Andrei Toom and further improved by Stephen Cook. There are several different versions of this algorithm, the most widely used is called: "Toom-3", although this name is also used as a collective name for all Toom-Cook based algorithms [11]. We will focus on the version Toom-3.
Likewise as in the algorithm of Karatsuba, the numbers to be multiplied are divided into parts; not in two parts, but in three parts.
This division is then applied recursively with the Toom-3 algorithm. This continues until the size of a core multiplier is reached, or until it is more efficient to use another algorithm (such as Classic or Karatsuba) for the last levels of recursion. The Toom-3 algorithm is much more complicated than the Karatsuba algorithm. We will discuss it only briefly.

In short, the Toom-3 algorithm works as follows:
Two numbers x and y are each split in three equal parts (equal number of bits). On the three parts of number x some simple formulas are applied, creating five new numbers. The same happens with the three parts of number y. The five numbers of x will be pairwise multiplied by the five numbers of y. The five products are then transformed, using again some simple formulas, to five numbers, which can be integrated to deliver the final product.
Where originally three times three multiplications would be required (according to the classical method) now with this algorithm it suffice to use five multiplications. This results in savings of 5/9. However, there are quite a lot of additions and subtractions and "small" multiplications or divisions needed. For example, multiplications by 2 or 3, or divisions by 2 or by 3. Therefore, only with larger numbers is Toom-3 faster than Classic or Karatsuba.
The complexity of the Toom-3 algorithm in big O notation is $O(n^{\log(5)/\log(3)})$ or approximately $O(n^{1.465})$.
Remark. In this document the terms Toom-Cook and Toom-3 are used interchangeably, but they always mean the same algorithm.

Schönhage-Strassen Algorithm (SSA)

In 1971 Volker Strassen and Arnold Schönhage came to the idea that a much greater level of D & C (Divide and Conquer) strategy was possible using the *convolution theorem.* Using this theorem, a number is divided in *n* parts (referred to as elements). Each element consists of an equal number of digits, or in computer terms, an equal number of bits. The last element is supplemented as necessary with zero bits. Such numbers can be regarded as a vector. On two of these vectors a *convolution* can be applied, which results in the product of these two numbers.[9] For large numbers, a significant acceleration can be achieved with the SSA algorithm. The complexity of the SSA algorithm is $O(n \log n (\log \log n))$.

Convolution theorem

Convolution is a mathematical operation on two functions with a new function (convolution) as a result. A number can be seen as a function (vector), and two numbers multiplied with each other can be seen as the result of a convolution.
According to the "convolution theorem", the (cyclic) convolution of two vectors *a* and *b* are found as follows:
Take the Discrete Fourier Transform (DFT) for each vector; multiply the resulting vectors in pairs, element by element (called the “Dyadic Stage” in this document); take the Inverse Discrete Fourier Transform IDFT of the products and integrate them to form the final product (cyclic convolution) *ab*.

First, let's discuss the DFT, then later convolution.

Discrete Fourier Transform

The Discrete Fourier Transform (DFT) is a mathematical treatment of a series of numbers, which particularly finds application in the digital signal processing. With the DFT a frequency spectrum of a digital signal can be determined. But DFT can also be used to carry out a discrete convolution.

When a DFT is applied to a series of numbers, then a new set of numbers is created with the same number as the original sequence. This process is called the "forward" process. The process is reversible, so that with an "inverse" operation, the original sequence can be obtained again.

The Discrete Fourier Transform is a laborious process, in most cases executed on a computer. For long sequences of numbers that require a lot of computer time. In 1965 James W. Cooley of IBM and John W. Tukey of Princeton published a method to carry out a DFT highly efficient on a computer; a method using "nested loops" or recursive called routines. This method is referred to as Fast Fourier Transform (FFT). In 1966, a similar method was published by W.M. Gentleman and G. Sande.

Fast Fourier Transform

When the FFT is used in the multiplication of numbers, they are divided into a number of equal parts, that is to say, parts with an equal number of decimal digits, or in computer terms, with an equal number of bits. These parts (called elements) then form the input sequence for the FFT. The number of elements of the input is referred to as N (but is sometimes also referred to as D) and must be preferably a power of two. The exponent of two is usually indicated by the letter k ($N = 2^k$). If the number of elements of the input array is smaller than a power of two, then the array is completed with elements having a value of zero up to the number of elements is a power of two.
With the Cooley-Tukey method, as well as with the Gentleman-Sande method, the elements of the numbers to be multiplied, are stored in computer memory in such a way that they, after processing, can be put back in the same place in memory. An operation is done on two elements at the same time, and consists of the determination of the sum and the difference of the two elements. The sum is returned to the original space of the first element and the difference is returned in the area of the second element. In this manner, a very memory-efficient process is obtained.
In a very particular order always two elements are taken and processed, then two following elements are treated. This continues until all elements have been processed. This is followed by a next round whereby a different composition of the two elements takes place. Thus, all the elements are processed once per round turn. The number of rounds is dependent on the number of elements $N (=2^k)$ and is equal to k. This process is performed separately for each of the two input numbers. Thereafter, the elements of the input numbers are multiplied pairwise with each other. The resulting products are then treated with a reverse procedure after which the elements are integrated into the final result.

When this calculation is schematically depicted then a form arises that is similar to the wings of a butterfly. Therefore, this calculation is called the "*butterfly*"[4]. For the forward *butterfly* the procedure of Gentleman-Sande is used and for the inverse *butterfly* the procedure of Cooley-Tukey.

There are many types of FFT algorithms developed. Often the input of an FFT is an array of complex numbers, which are stored in the form of floating point numbers. However, for the multiplication of large numbers according to the SSA algorithm, it is obvious to use integer numbers. For SSA the most appropriate form of FFT is the *Number-theoretic transform*.

Number-theoretic transform (NTT)

The Number-theoretic transform is a type of Fast Fourier Transform that works with integer numbers in the domain modulo a number [5]. This number (the modulus) can be a prime number, but that is not essential.
A special form of NTT is the Fermat transform modulo $2^n +1$. (The term "Fermat transform" is somewhat misleading. The modulus of a Fermat **transform** does have the form $2^n +1$, but the exponent n is not necessarily a power of two. But with a Fermat **number** (2^n+1) the exponent n is absolutely a power of two). [3]
So this form of NTT takes place in the field of integers modulo ($2^n +1$) and is generally used in the SSA algorithm. The formula for this kind of NTT is:

$$X_j \equiv \sum_{i=0}^{N-1} x_i g^{ij} (mod\, m)$$

In this formula, the array of N numbers x_i is transformed into the array X_j, also consisting of N numbers, where both i and j range from 0 to $N-1$. The array of x_i can be seen as an input while the array X_j can be considered as output. Furthermore, g is the n-th root of unity modulo m, where m is a number of the form: 2^n+1. The numbers of the output array X_j are all of the form (mod m).

Roots of unity

The nth root of unity is the number that, raised to the n-th power, 1 as result has [8]. Usually, that is a complex number. For example i (= √ -1) is the 4th root of unity, for $i^4 = 1$. Of course 1 can also be a root of unity, but 1 as a root is in mathematics not interesting. Therefore, the concept of *primitive root of unity* is introduced; an n-th root of unity x is primitive if $x^n = 1$ and all powers < n do not result in 1.

Also in the domain of numbers modulo m, the concept of root of unity is known[7].
A k-th root of unity modulo m is the number x that raised to the k-th power (mod m) has 1 as result. A k-th root of unity x is primitive if $x^k \equiv 1$ (mod m) and any power smaller than k does not yield 1 (mod m).

Fourier transform according to the Number-Theoretic Transform

If we analyze the formula for the NTT we see as "input" a row of N elements x, and as "output" a row of N elements X , where $N = 2^k$. Each element of X (X_j) is formed by the sum of the elements x, multiplied with a constant (*the n-th root of unity*) to a certain power ij. We can now depict the formula in the form of a table, in which the columns are formed by the "input" and the rows by the "output". As an example, we take a string of N elements, with $N=2^k$, where k=3, so $N=2^3=8$. We can put the exponents formed by $i\,j$ in a table first:

Input i =	0	1	2	3	4	5	6	7
Output: j = 0	0	0	0	0	0	0	0	0
1	0	1	2	3	4	5	6	7
2	0	2	4	6	8	10	12	14
3	0	3	6	9	12	15	18	21
4	0	4	8	12	16	20	24	28
5	0	5	10	15	20	25	30	35
6	0	6	12	18	24	30	36	42
7	0	7	14	21	28	35	42	49

Table 1

We now take $2^8+1 = 257$ as modulus, then the 8th root of unity = 4, for $4^8 \equiv 1 \pmod{257}$. We can now apply the exponents from table 1 on the root of unity 4 (take into account modulo 257) and get the results in table 2:

Input i =	0	1	2	3	4	5	6	7
Output: j = 0	1	1	1	1	1	1	1	1
1	1	4	16	64	256	253	241	193
2	1	16	256	241	1	16	256	241
3	1	64	241	4	256	193	16	253
4	1	256	1	256	1	256	1	256
5	1	253	16	193	256	4	241	64
6	1	241	256	16	1	241	256	16
7	1	193	241	253	256	64	16	4

Table 2

Because all values are (mod 257) the values can also be taken negative: 256 becomes -1; 253 becomes -4; 241 becomes -16 and 193 becomes -64. The result we see in table 3:

Input i =	0	1	2	3	4	5	6	7
Output: j = 0	1	1	1	1	1	1	1	1
1	1	4	16	64	-1	-4	-16	-64
2	1	16	-1	-16	1	16	-1	-16
3	1	64	-16	4	-1	-64	16	-4
4	1	-1	1	-1	1	-1	1	-1
5	1	-4	16	-64	-1	4	-16	64
6	1	-16	-1	16	1	-16	-1	16
7	1	-64	-16	-4	-1	64	16	4

Table 3

The values 1 and -1,   4 and -4,   16 and -16 and 64 and -64 remains. These values are called “twiddle factors”, and with these values the elements of the input are multiplied.
We assume that the input consist of a number that is divided in up to 8 equal parts (equal number of decimals or equal number of bits), which forms the input for the columns 0 to 7.
When we now multiply the input value of column 0 with all the twiddle factors of column 0 and do so for all the columns, we perform a complete Fourier Transform by hand.
The sum of the products of the rows of the table forms the output.

We can now transfer the output of row 0 to the input of column 0 of a new table, with the same “twiddle factors”, transfer output of row 1 to input of column 1 etc. and apply on this new input the *Inverse Number Theoretic Transform,* then we get the original input again.
However the order of the elements is changed; the content of column 0 stays in column 0, but what originally was the content of column 1,2,3,4,5,6,7 is now the content of column 7,6,5,4,3,2,1.

The formula for the *Inverse Number Theoretic Transform* is:

$$x_i \equiv \frac{1}{N} \sum_{j=0}^{N-1} X_j g^{ij} (mod\, m)$$

The formulas for *NTT* and the *Inverse NTT* are almost identical; the difference is that with the inverse form the elements must be divided by *N (mod m)*. All the numbers are *(mod m)*.

Fast Fourier Transform

When we take a precise look at the table of the example we can observe some symmetries; the left four columns contain the same factors as the right four columns; only the signs can differ. For the even numbered rows the signs are equal but for the odd numbered rows the signs are opposite.

This can be used cleverly by calculating the sum and the difference of the values of column 0 and column 4; the sum is then applied to the even-numbered rows and the difference is usable for the odd-numbered rows.

We imagine a ***container*** in which there is room for *N* elements (in the example, 8 elements, numbered 0 to 7). Each element can contain a value that is limited to the modulus; in our example the modulus used is $2^8+1 = 257$, so the value in each element is limited up to and including 256. We begin with the placing of the input values in the elements of the container. We now take column 0 and column 4, (or we better can speak of element 0, and element 4 of the container), we determine the sum of element 0 and element 4 and place the sum back in element 0; we determine the difference of element 0 and element 4 and place the difference back in element 4. Before or after calculating the difference we need to multiply the element with the twiddle factor (more on that later).

In the same way the sum and the difference for elements 1 and 5 can be determined. Similarly, for element 2 and 6, and for element 3 and 7. The container now contains eight elements with a content completely different from the content at the beginning.

The process may now be repeated with this new content, but then element 0 coupled to element 2, element 1 is coupled to element 3, 4 is coupled to 6 and 5 to 7.
Now in the last tour element 0 and 1 are taken together; element 2 with 3, 4 with 5 and 6 with element 7. Instead of eight times computing 8 elements, only k times ($k = 3$) calculating the new value of 8 elements will do the job. This is a considerable acceleration by the principle of the Fast Fourier Transform.

Twiddle Factors

The twiddle factors in Table 3 illustrate the type of factors and where these factors are applied. But if the rows of the table are merged during the FFT procedure the factors are not right anymore . Thanks to the intelligent algorithms of Cooley-Tukey and Gentleman-Sande these factors are easily calculated and applied (see example worked out in C^{++} code) .
As was seen in the example in Table 3, with $N = 8$, there are four different twiddle factors. It turns out that for any value of N the number of distinct twiddle factors is equal to $N / 2$.
The first factor should be: a primitive root of unity of order N. Thus, when $k = 3$ and $N = 2^k = 8$ the first twiddle factor should be a primitive 8th root of unity. With integers in the domain (mod 2^n+1) all twiddle factors are powers of two.
When $k = 3$, the first factor is $2^{(n/4)}$, the following factors are $2^{(2n/4)}$ and $2^{(3n/4)}$. The last twiddlefactor is 2^n. All these factors are modulus 2^n+1. Thus, the last factor has the value $2^n (\text{mod } 2^n+1) = -1$.

(The capital letter N is used herein to indicate the number of elements; the lower case letter n is used as an exponent of 2 in order to express the size of a number (as is the case in the modulus $2^n +1$). In addition n is used to express the number of bits that is required to store a number of up to 2^n .)

Convolutions

We have seen now that a number can be divided into elements and how on these elements a Discrete Fourier Transform can be applied. If we want to multiply two numbers *a* and *b*, it can be done according to the convolution theorem, by calculating for both *a* and *b* the Discrete Fourier Transform (DFT), multiply the elements of these DFT's of *a* and *b* in pairs with each other and again apply to these products the Inverse Discrete Fourier Transform (IDFT). The results are then merged into the end result: *ab* . Schematically it looks like this:

Product *ab* = IDFT (DFT (*a*) * DFT (*b*)) [2]

In the SSA algorithm the DFT is carried out as a Number-theoretic transform (NTT), so we can, in this case, replace the terms IDFT and DFT by INTT (Inverse NTT) and NTT. The NTT and INTT are executed as a Fast Fourier Tranform (FTT).

Formally the SSA algorithm promises us the product ab (mod 2^n+1), if the input numbers *a* and *b* both are $\leq 2^n+1$ [13]. We can translate this as follows: if the input numbers are not greater than *n* bits, the SSA algorithm gives us the product of these numbers mod (2^n+1) . The product of two numbers with a maximum of *n* bits results in a number of *2n* bits at maximum. We are obviously not interested in the product *ab* if it is limited to *n* bits (which is the case when mod (2^n+1) is used). To get the entire product we have to choose *n* so large that the entire product $ab < 2^n+1$. The modulus used here is different from the modulus used in the Number-Theoretic Transform !

There are various forms of convolution which yield different results [2]:

The ***acyclic convolution*** gives us the full product *ab*. We achieve this by choosing *n* so large that $ab < 2^n+1$. This can be accomplished by choosing the right size and number of elements in the FFT process. Let *N* be the number of elements of the container for the FFT; then, at the beginning of the FFT process, we fill only half of the elements *(N/2)* with inputdata and set the remaining elements to zero. This is called *"Zero padding"*.
We have to ascertain that the size of the elements is sufficient to contain the full input. (See section Content).

The ***cyclic convolution*** we get if we fill all the elements of the container. The result is $ab \pmod{2^n-1}$. An example of c*yclic convolution* we see later . This is also called a "Mersenne Transform"', because of the form of the modulus. In some implementations, this transformation is used in SSA[3]. We do not go on this further.

The ***negacyclic convolution*** returns as result $ab \pmod{2^n+1}$ and this is the convolution formally used in the SSA algorithm. We obtain this result by "weighting" the input elements (multiplying with a weighting factor) before the forward butterfly and multiply the result of the FFT after the inverse butterfly again with an inverse weighting factor. This process is called DWT (Discrete Weighted Transform) and the original SSA algorithm [2] uses DWT . The reason why the SSA algorithm makes use of the negacyclic convolution is to be able to apply the algorithm recursively. During the "Dyadic Stage", the contents of the elements of

container *a* are multiplied pairwise with the content of the elements of container *b.* Then on the product the modulus (mod 2^n+1) is applied. When we use for the Dyadic Stage recursively the SSA algorthm with negacyclic convolution, then the lower level and higher level recursions fits nicely together (both mod 2^n+1).

In practice, the SSA algorithm is not always used recursively and with the negacyclische convolution. At the highest level it is in any case necessary to use the acyclic convolution in order to obtain the entire product. If SSA is used recursively this is limited to only 1 or 2 levels deep[13].

Content

The elements of a container are designed so that they can contain any possible value modulus *(2^n+1)*. The smallest value is zero, and the greatest value is 2^n. At the start of the FFT process, the elements are filled with input. But not every value is allowed because during the process the content of some elements are added together. Therefore, there is a limitation to the value with which the elements can be filled initially.
We take for example numbers modulus ($2^{16}+1$). The smallest value is zero, the maximum value is 65,536 . Each of the different values between the smallest and the greatest can be stored in a binary 16-bit field (except 2^{16} = 65,536 , but more about that later).
Let's assume k = 3 as we did with table 1,2, and 3. Then we get a table with 8 lines whose elements eventually have to be added together. Of the 16 bits of the binary field we therefore need 3 bits ($8=2^3$) to be reserved for this summations . The number of bits remaining should be divided by two because of the pair-wise multiplication. So 13 bits divided by two is rounded down 6 bits.

The maximum value by which the elements of the containers can be filled at the start of the FFT process is, in our example, a number of 6 bits, or values between 0 and 63 inclusive. This value is in the literature often referred to as M; however in this document the term *"Content"* is used .
Content is therefore the number of bits with which the elements at the beginning of the FFT process can be filled. So the input-number (as string of bits) is divided in parts of *Content* bits long.

The SSA algorithm step by step

We now have enough information to outline the entire process of the SSA algorithm. We have two (large) numbers *a* and *b* that we want to multiply so that we get the product $c = a * b$. We assume that the numbers *a* and *b* have the same order of magnitude. On the basis of this size, we define a *k*, a number of elements $N=2^k$, and a modulus *m* (later we will see how to do that). Furthermore, we create two containers, one for *a* and one for *b*. The containers each contain a number of *N* elements, each element can contain a value (mod *m*).

Since we apply the acyclic convolution we split up *a* into *N/2* equal parts. Each part contains an equal number of bits, that number of bits is equal to the *Content.* We fill the elements of container *a* with *"content"* bits; we use only the first half of the elements; remaining elements are filled with zero. We do the same with number *b* and container *b.* (The containers are of equal size).

We now are going to perform for container *a* the FFT according to the Gentleman-Sande method; the twiddle factors are calculated and applied and the elements are added and subtracted. All operations (addition,subtraction and multiplication with the twiddle factor) happen *(mod m)*, so that the result always fits in an element of the container. Now the elements in container *a* have been given a changed content. In a similar manner, the FFT is performed on the elements of container *b*.
Next, the elements of container *a* are multiplied in pairs with the elements of container *b*. So element 0 of container *a* is multiplied by element 0 of container *b*, and so on. We will call this step the "Dyadic Stage". For these multiplications we can use the Classical method, or the method of Karatsuba or Toom3. But we can also use the SSA algorithm (in our example in C^{++} we will use the Classical method).
The results are *(mod m)* placed in the elements of container *c.* (We call the container in which the result of the Dyadic Stage is put container *c*. Actually, this is either container *a* or container *b*, which one does not matter).
The next step is to perform the inverse FFT according to the scheme of Cooley-Tukey. The results are again placed back in the elements of container *c*. As a final step, the contents of the elements are divided by *N(mod m)*. The container now contains the *acyclic convolution* of number *a* and number *b*. The elements of container *c* are now being put in the right order and integrated into the final result $c = a * b$.

Example

As an example we take the multiplication-example of the classical method. Number *a* = 1234 and number *b* = 5678. We create two containers *a* and *b* each with 8 elements and use as modulus $2^{16}+1$ = 65537. In this example we place in each element only one decimal digit. (The maximum *Content* for a modulus $2^{16}+1$ is 6 bits or the values 0 to 63 inclusive. We use here however as *Content* the value of 10; so each element can contain one decimal digit, or values 0 to 9 inclusive).
We are now filling the elements of container *a* with the digits of the first number: 4 in element 0, 3 in element 1, 2 in element 2 and 1 in element 3. The elements 4 to 7 we fill with zero, thus we perform the ***acyclic convolution***, so that we can achieve a correct result.

element :	0	1	2	3	4	5	6	7
	4	**3**	**2**	**1**	**0**	**0**	**0**	**0**

Container a

The elements 0,1,2,3 of container *b* we fill with the digits of the second number, respectively 8,7,6,5 and elements 4 to7 we fill with zero.

element:	0	1	2	3	4	5	6	7
	8	**7**	**6**	**5**	**0**	**0**	**0**	**0**

Container b

We now execute the convolution as described above. The result comes in the elements of container *c*, (in reality it is container *a*) which now contains:

element :	0	1	2	3	4	5	6	7	8
	32	**0**	**5**	**16**	**34**	**60**	**61**	**52**	**32**

Container a after the convolution

The elements are now put in the right order; for example by moving element 0 to a virtual element 8. Now the elements 8 to 1 represent in descending order the ascending values 1, 10, 100, 1000 etc. This is exactly equal to the example we used in the classical method.
The content of the elements is now multiplied by the corresponding power of ten (just as we saw in the example of the classical method):

```
Element 0   32 x   10^0 =         32
Element 7   52 x   10^1 =        520
Element 6   61 x   10^2 =      6,100
Element 5   60 x   10^3 =     60,000
Element 4   34 x   10^4 =    340,000
Element 3   16 x   10^5 =  1,600,000
Element 2    5 x   10^6 =  5,000,000
                           ----------
         1234 x 5678 =     7,006,652
```

The columns in the container apparently represent each a certain value, which is a power of ten. It is the *convolution process* that ensures that the numbers end up in the right element. It should be noted that the elements in the container have a much greater capacity than 0 to 9. In the classic school method we ensure that a "carry" is counted in the column with the higher power of ten directly, but at the SSA method we just count in one element and we make sure that "carries" are processed later.
In this example we have filled the container but half full. What happens when we fill container *a* on? We take the test and add a 1 in element four of container *a* and then carry out the convolution again. The result is:

element:	0	1	2	3	4	5	6	7	8
	32	**5**	**11**	**23**	**42**	**60**	**61**	**52**	**32**

Container a after the convolution of 11,234 x 5,678

We can see that in element four a value of 8 is added; in element three a value of 7; in element two a value of 6 and in element one is a value of 5 added. That's exactly what we can expect as the 1 of element four in container *a* represents a value of 10,000, which multiplied by 5678 gives an addition of respectively. 5, 6, 7 and 8 in the resp. elements one,two,three and four. Element one represents the highest value: 10 million or 10^7 and is now filled (with 5).

What happens when we then add 4 to element four of container *b* ? Element four represents a value of 10,000 and when we multiply a 4 of this element of container *b* by the 1 of element four of container *a* then the result is 4 times 10,000 times 10,000 or $4x10^8$. There is in the containers, however, not an element that represents the value 10^8!
If we redo the convolution we see the result:

element:	0	1	2	3	4	5	6	7	8
	36	**9**	**19**	**35**	**58**	**60**	**61**	**52**	**36**

Container *a* after the convolution of 11,234 x 45,678

element 0 (8)	36	x	1	=	36		
element 7	52	x	10	=	520		
element 6	61	x	100	=	6,100		
element 5	60	x	1,000	=	60,000		
element 4	58	x	10,000	=	580,000		
element 3	35	x	100,000	=	3,500,000		
element 2	19	x	1,000,000	=	19,000,000		
element 1	9	x	10,000,000	=	90,000,000		
					113,146,656	(mod 99.999.999)	**13,146,657**
	11,234	x	45,678	=	513,146,652	(mod 99.999.999)	**13,146,657**

In element four is 4 x 4 = 16 added; 4 x 3 = 12 is added to element three, 4 x 2 = 8 is added to element two and 4 x 1 is added to element one. Then there still is 4 x 1 = 4 (the 1, which has been previously added to container *a*) , that would have to be added to an element which the value 10^8 represents, but that element does not exist. The 4 is just added to element zero. This is called "wraparound". For values that do not fit regular in the container, the counting is just started again at element zero and the new content will be merged with the existing content! The result of this convolution is thus **not** equal to the result of the ordinary multiplication, but if the results are taken (mod 99,999,999) they are indeed equal to each other. Mod 99,999,999 is not taken arbitrarily, this modulus equals ten to the power of eight (the number of elements of the container) minus one. Thus (mod 10^8-1).
So this is the example of the ***cyclic convolution***.

If we want to show an example of the ***negacyclic convolution***, then we have to multiply the input with a weighting factor. This weighting factor is different for each element of the container and depends on the number of elements in the container N and the size of the modulus: the n in mod (2^n+1). Specifically, the weighting factor is an exponent of two. The base value for the exponent is n/N and the exponent is multiplied by j as the index for the elements.
In our example, $N=8;\ n=16$ (from the modulus 2^n+1), and the index j is from 0 to 7 inclusive. The value n/N =16/8 = 2 and the exponents are: 0,2,4,6,8,10,12 and 14. The weighting factors are thus 2^0, 2^2, 2^4, etc. to 2^{14}. The input of the elements 0 to 7 is multiplied by resp. 1,4,16,64,256,1024,4096 and 16392, all mod (2^{16}+1). After the weighting is applied the input will look like this:

element :	0	1	2	3	4	5	6	7
	4	**12**	**32**	**64**	**256**	**0**	**0**	**0**

Container *a*, input after multiplication with weighting factor

element:	0	1	2	3	4	5	6	7
	8	**28**	**96**	**320**	**1024**	**0**	**0**	**0**

Container *b*, input after multiplication with weighting factor

With the weighted input the Number-theoretic transform is performed. The elements of container *a* and *b* are multiplied by each other. Then the inverse Number-theoretic transform is executed and the output is placed in container *c* (actually this is container *a* but it can also be container *b*, that does not matter) and looks like this:

element:	0	1	2	3	4	5	6	7	8
	224	**65,519**	**32,759**	**24,572**	**53,247**	**30,720**	**7,808**	**1,664**	**224**

Container *a* after the negacyclic convolution of 11,234 x 45,678 before weighting

The order of the elements is changed by the transformation: element 0 still represents the value 1, but element 7 now represents the value 10, element 6 represents the value 100 etc. By moving the content of element 0 to the virtual element 8, the elements 8 to 1 in descending order represent the values 1 to 100,0000,000 in ascending order.

The output must still be "weighted" by dividing the content of the elements by the same factor as was used to multiply the input. In addition, the output also has to be divided by a factor *N* (see the formula for the inverse NTT). Furthermore, the division has to happen *mod* (2^n+1)!

Modular division is, in terms of computer cycles, an expensive process. Fortunately, there is a trick that makes division simpler, when the divider consists of two to a certain power, and the modulus is of the form (2^n+1). The division can be converted into a multiplication by multiplying the exponent of the modulus by two and decreasing the result by the exponent of the divisor. What remains is the exponent of two; now two to the power of that exponent is the factor to be used as multiplication factor.

In our example, the divisors are equal to the weighting factors: for the elements 8 to 1 are that resp. $2^0,2^2,2^4,2^6,2^8,2^{10},2^{12},2^{14}$. The exponent of the modulus times two is 32, which now must be reduced by resp. 0,2,4,6,8,10,12 and 14 . The division by *N* can happen together with the division by the weighting factors. So the 32 (two times the exponent of the modulus) must be reduced by an additional 3 $(N=2^3)$. The resulting exponents for the elements 8 to 1 are respectively 29,27,25,23,21,19,17 and 15.

The value of the elements must now be multiplied by resp. $2^{29},2^{27},2^{25},2^{23},2^{21},2^{19},2^{17}$ and 2^{15} and all that $(mod\ 2^{16}+1)$. The result is:

element:	0	1	2	3	4	5	6	7	8
	28	**9**	**19**	**35**	**58**	**60**	**61**	**52**	**28**

Container *a* after the negacyclic convolution of 11,234 x 45,678, after weighting

element 0	28	x	1	=	28		
element 7	52	x	10	=	520		
element 6	61	x	100	=	6,100		
element 5	60	x	1,000	=	60,000		
element 4	58	x	10,000	=	580,000		
element 3	35	x	100,000	=	3,500,000		
element 2	19	x	1,000,000	=	19,000,000		
element 1	9	x	10,000,000	=	90,000,000		
					113,146,648	(mod 100.000.001)=	**13,146,647**
	11,234	x	45,678	=	513,146,652	(mod 100.000.001)=	**13,146,647**

The result of this convolution *(mod* 10^8+1*)* is equal to the result of the multiplication of 11,234 x 45,678 *(mod* 10^8+1*)*. So, this is the example of the ***negacyclic convolution***.
The story of negacyclische convolution is not finished yet. Under some circumstances, there is still a correction to take place on the weighting: if the contents of an element after the weighting has a larger value than is theoretically possible, then the contents must be corrected (reduced) by the modulus, so is to be reduced by (2^n+1). This may result in a negative value for the content of an element. In the implementation this should be taken into account.

The theoretical maximum value that an element can contain is different for each element and is determined as follows: the index *j* of an element is increased by one and then multiplied by the C*ontent* of an element squared (in our example 10^2, but in practice *Content* is a number of bits, which represents the value 2^{Content}.)
The index *j* is the index of an element that occurs after the inversion by the FFT process, and is therefore in our example: $j = 0$ for element 8, and so on until $j = 7$ for element 1.

Two moduli are mentioned here, what can lead to confusion; modulus (2^n+1) in the formula of the NTT and modulus (10^8+1) that is applied to the final result of the negacyclic convolution and further in this document wil be called "*modulusN"*.
These are two different moduli. However, if the negacyclic convolution is applied in the Dyadic Stage, then it is the intention that both moduli coincide and become equal to each other.

The size of containers and elements

In a multiplication of two different numbers, we use two containers, *a* and *b* , which should be of the same size. (If we want to calculate the square of a number then one container will suffice.) The size of a container is determined by the number of elements in a container and the size of an element. The number of elements N is given by $N = 2^k$. The size of an element is dependent on the modulus that is used . The modulus is 2^n+1 (n is different from N). The maximum value that an element must be able to contain is modulus-1 is thus 2^n. An element that is composed of n bits can contain at maximum the value 2^n-1 and that is sufficient as long as we have a solution for the (rare) cases that the value 2^n must be stored.

At the start of the forward butterfly procedure the containers are loaded with the numbers to multiply. Those numbers should be split into parts, how many depends on the convolution that we want to apply . In the acyclic convolution we can only fill half the elements. So we need to split the numbers up to $N/2$ parts . The remaining elements are filled with zeros (zero padding) .
This splitting of a number can in different ways; we can divide decimal numbers as described in the previous example. The columns (elements) then get resp. the values 1,10,100 .. etc. At composing the final result, the contents of elements are then multiplied by the corresponding value (1,10,100 .. etc.).

In a computer program numbers are usually stored as a row of bits . We can split up such a number in parts with the same number of bits. We can, for example, divide a number in parts of 32 bits. The value of the elements is then, respectively 1, 2^{32} ,2^{64}, .. etc. The number of bits that we can fill the elements with at the start of the forward butterfly process is limited to *Content*, as we have seen in the section Content. The effective *Content* is at maximum $(n-k)/2$ bits for an element with modulus 2^n+1, but may be less .

Working with numbers of bits, that is not a multiple of eight, is in most computers rather inefficient. Therefore *Content* can best be rounded down to a multiple of eight (bits). At multiples of eight bits the program can work with full bytes. In the implementation discussed here is chosen to round to a multiple of 32 bits. As a result, the input numbers can be processed in multiples of a machine word of 32 bits.

The choice of *k*

The value of *k* plays an important role in determining the size of the container and the effective content that can be placed in it (that is the input). The number of elements of a container is 2^k. Of these, with the acyclic convolution, only half is usable at the start of the FFT; so $2^k/2$ elements.
The minimum size of an element is $2^k/2$ bits to accommodate the number of different twiddle factors (see twiddle factors). There are however tricks to accommodate more twidde factors, but that is beyond the scope of this document[3].
The maximum *Content* of an element at start of the FFT is $(2^k/2-k)/2$ bits (see previous section). The maximum content of the entire container at start of FFT is: $2^k/2$ elements times $(2^k/2-k)/2$ bits. If *k* is one larger, then the number of elements and the *Content* of the elements increases with (almost) a factor of two. (Almost since the *Content* of an element is decreased with the factor k). Thus, the capacity of the container as a whole (in number of bits) will increase by a factor of approximately four!
We can also increase only the size of the elements of the container by a factor of two. This increases the total capacity of the container by a factor of two.
If the factor *k* is known, we can calculate the capacity of the container. If we do not know *k*, but we do know the capacity we need to store the input, we can only make a rough estimate of the factor *k*. In practice, the optimal value of *k* can best be determined experimentally. That is what we are going to do when we test our SSA implementation.

Acyclic and negacyclic convolution compared

The acyclic convolution provides the complete product *ab* of integers *a* and *b*. The negacyclic convolution provides only the product $ab \bmod (2^n+1)$. It is obvious to use the acyclic convolution at the top level of the SSA algorithm. The negacyclic convolution can be applied at the lower level (during the Dyadic Stage). Yet it is not certain that at the lower level the negacyclic convolution is a better solution than the acyclic convolution.
The advantages and disadvantages of the two convolutions are listed here:

Acyclic convolution	*Negacyclic convolution*
The result of this convolution is the complete product *ab* of both factors *a and b*.	The result of this convolution is the product $ab \bmod (2^n+1)$ of both factors *a* and *b*. This convolution does not provide the complete product.
Only half of the elements of the container can be filled with input-data. So more elements or bigger elements are required.	All elements of the container can be filled with input-data.
The size of an element must be divisible by half *K*; so divisible by $2^{(k-1)}$. This makes the choice of a suitable size of element for the container more flexible.	The size of an element must be divisible by *K*: so divisible by 2^k.
No weighting is required.	The elements are to be multiplied with a weighting factor.
After the backward FFT the elements of the container must be divided by $N \bmod (2^n+1)$. However, when *content* is limited to: *(content in bits)/2 – k*, then application of modulus (2^n+1) can be omitted. Only a division by *N* is needed; this can be accomplished by a simple shift-right over *k* bits	After the backward FFT the elements of the container must be divided not only by $N \bmod (2^n+1)$ but also by the weighting factor $\bmod (2^n+1)$. These two divisions can be converted into one multiplication $\bmod (2^n+1)$. Additional a check must be made that the value of the element not is greater than the theoretical maximum. If it is then the value must be decreased by the modulus (2^n+1). The value of the element can then be negative and additional measures are needed.

In summary, the negacyclic convolution is considerably more complex than the acyclic convolution, but makes it possible to fill twice as many elements. Tests will demonstrate the advantages in terms of speed in applying the negacyclic convolution during theDyadic Stage.

The Implementations

In order to compare the different algorithms in terms of execution time, we need to have available the implementations of these algorithms. In the LBA-library (see the next section) we already have available a routine for squaring via the classical method, a routine for squaring according to the Karatsuba algorithm; and for squaring according to the Toom-Cook algorithm. All these routines are designed for a 32-bit Intel architecture in a Windows environment. It is most practical to implement also for the SSA algorithm a squaring routine for a 32-bit Windows environment.
Squaring is a limitation with respect to multiplication. Yet squaring occurs in many applications and is an important part of all the multiplications[13]. Therefore, it is justifiable to perform comparisons specific for squaring. The routines are written in assembly language but are callable from a C^{++} environment as well as from an assembler environment.

The implementation of the Classical algorithm

The implementation of a classical multiplication is a fairly simple routine. In two nested loops a computer word of the multiplicand is multiplied by all the words of the multiplier and the results are added in the output field. Next the following computer word of the multiplicand is handled and so on untill all the words are done. All this taking into account the positions where to count the products in the output field.
In the LBA-library, this function is available under the name “Mul”. There is in this library also a separate function present for squaring called “Smul” (Square Multiplication).

The implementation of the Karatsuba algorithm

There are two Karatsuba routines in the LBA-library available: “Karat” for multiplying two numbers and “Skarat” for squaring a number. The Karatsuba algorithm uses recursion. With “Skarat” the input is on every recursion level divided into two parts. With these two parts (numbers) a third number is calculated via simple formulas. For each of the three numbers a Karatsuba routine is called again (recursively), the routine “Skarat” for the two original numbers, and the routine “Karat” for the calculated number. This continues until the numbers are so small that it is more efficient to use the Classical algorithm.
The input consists of a number of computer words. In the ideal case this number of computer words is a power of two. Usually, the input does not comply with the ideal situation. Therefore it was decided to dynamically create a work field that has the ideal shape and wherein the input number is copied. Also dynamically an output field is created, twice as large as the input field; and intermediate work fields, in total five times the size of the inputfield.
The total space required for the fields is eight times the size of the input in computer words of 32 bits, rounded up to a power of two. Of those eight, one is for the input field, two for the output field and five for the intermediate results for all levels of recursion.
The product is copied from the output field to the external result field.

Implementation of Toom-Cook algorithm

For the Toom-Cook algorithm there is only an implementation for squaring in the LBA-library. This implementation uses also recursion. The input is divided into three parts each. On this three parts some calculations are applied which result in five new numbers. Each of these five numbers is squared by the Toom-Cook routine again (recursively). On the five results of the squarings some calculations are applied, and these numbers are integrated to form the final result[11].
The recursive calls continue until the numbers are so small that they better can be treated by the Karatsuba algorithm. This limit is set at 32 words of 32 bits, so 1024 bits.
To achieve a smooth transition between Toom-Cook and Karatsuba, the (optimal) input should have a size of 1024 bits (or 32 words of 32 bits) times a power of three. Dynamically some internal fields are created (as with the Karatsuba implementation): an input field equal to the size of the optimal input, an output field twice as large as the input field, and fields for the intermediate results in the amount of three times the optimal input size. This is for all levels of recursion together. The input number is copied to the internal input field: the end result is copied from the internal output field to the external output field.

The implementation of the SSA algorithm

For the SSA algorithm two implementations are made. One, written in C^{++} , is intended to test the principles of SSA and Fast Fourier Transform (FTT) and to demonstrate this principles. The second implementation, written in assembler language, is intended to be used, inter alia, in the comparison between the different algorithms. Both implementations calculate the square of the input number.

The C^{++} implementation of the SSA algorithm

The listing of the C^{++} implementation (at least the relevant part) is included in this document (at the end). The C^{++} implementation comes in two variants: a simple one shows the acyclic convolution which delivers the complete square of the input number; a more complex one shows the negacyclic convolution; this variant delivers the square of the input number mod *modulusN*.
In this implementations, the factor k, and the size of the elements are specified and fixed. The container has a fixed structure and has a limited number of elements with a fixed size. The forward FFT and inverse FFT are carried out in the form of a number of nested loops in accordance with the algorithms of resp. Gentleman-Sande and Cooley-Tukey. For the "Dyadic Stage", the classical method of multiplying by the routine “Smul” (Square multiply) from the Lba library is used.
The input is generated on the basis of a "requested number of bits", and consists of a random binary string of ones and zero's. For comparison, the input is also processed by the routine “Smul”. The result of “Smul” should be equal to the output of the SSA routine. In the negacyclic variant the output numbers of SSA and “Smul” should be equal mod *modulusN.*

The Assembler implementation of the SSA algorithm

For the assembler routine the emphasis is on execution speed. The routine is added to the Lba library and accepts an arbitrary number in the form of a binary bit string as input. The routine is designed to square numbers up to a size that just fit into the address space of Windows 32-bit version. This sets a limit to the factor k.

A value of 17 for k would mean for N a number of 2^{17} = 131,072 elements. The minimum size of an element would then have to be N/2 = 65,536 bits or 8,192 bytes, in order to accommodate all the different twiddle factors. (There is a way to use smaller elements by using “the √2 trick”[3]; but that is beyond the scope of this document). Total space required for the container would amount to at least 131,072 * 8,192 bytes = one GB (Gigabyte). In addition, space is required for the input and output fields. This is too much for the 2GB maximum address space of 32-bit Windows. Therefore, we chose a value of k that is at maximum 16 and at least 3.
In contrast to the implementations of Karatsuba and Toom-Cook the input for the SSA implementation is not copied to an internal field. There is only one internal field and that is the container. (In squaring there is only one input number, so also only one container).
The number of significant bits of the input number is determined and on the basis thereof,

the factor k is set. The corresponding number of elements and their size (expressed in number of bits), as well as the modulus are dynamically calculated and the container is created. The size of the container (the amount of storage space occupied by the container) is about four times the size of the input. The maximum content per element is calculated , and the container is filled with the input. The elements that are not used are filled with zero.
At the highest level, the acyclic convolution is applied so that only half of the elements is filled. In the determination of the value of k and the size of the elements this is taken into account.
For the "Dyadic Stage", in which the contents of the elements is squared, the classical algorithm was elected for the smaller elements. For larger elements the Karatsuba routine for squaring (Skarat) is used. The boundary is placed at 4,096 bits.
With elements of 10,240 bits and bigger the SSA algorithm is used in the "Dyadic Stage". This does **not** happen recursively because for the second level SSA the negacyclic convolution will be used and it was much easier to develop a separate routine for the second level SSA algorithm than making the first level re-enterable.

The optimum value of k for the second level SSA is experimentally determined.

For the second level "Dyadic Stage" only the Classical algorithm is used. So, there is no third level SSA algorithm.

Technical aspects

In the C^{++} implementation routines are used from the Lba function library. These routines work with large positive numbers and zero. With negacyclic convolutions the contents of elements may become negative. To prevent that, the possible reduction of the content of an element with modulus (2^n+1) is not done directly, but the reduction is done in the total result field. To prevent that this field becomes negative the result field is initialized with the value of *modulusN*. At the end of the calculation, *modulusN* is applied to the result-field to ensure that the initial value, if necessary, is compensated.

The assembler implementation also make use of Lba functions. The elements of the container are constructed as LBA-fields. An LBA-field consists of a length field and a number of limbs (the term limb is typically used with software that calculate with big numbers; it indicates that a limb is part of a larger whole). In this case a limb is a computerword of 32 bits. An element of the container always consists of a number of full limbs, the minimum size is one limb of 32 bits. The length field is also a computer word of 32 bits, it contains the number of limbs of the LBA field..
The content of the elements is limited by the modulus, and thus can not be more than or equal to the modulus, which is in the form of 2^n+1. The size of an element of the container will be expressed as n bits. This n is divisible by 32. The maximum binary value in a computer field of n bits is 2^n-1 . So a solution had to be found for the (rare) situation that the contents of an element is exactly the value of 2^n.
The length field of an LBA field has 32 bits of which 30 bits are used to indicate the number of limbs, (this number can in the 32-bit version of Windows never be more than 2^{30} (about 1 billion). Thus, there remain two bits that can be used by the SSA implementation; one bit to indicate that the content has the value 2^n, and one bit is used to indicate that the content is equal to zero.

In the same manner as in the C^{++} routine is the result-field for the negacyclic convolution initialized with *modulusN.* This is to avoid that the result field becomes negative.

The *Content* per element (that is the initial value by which the elements can be filled at the beginning of the FFT transform) will always be rounded up to whole limbs of 32 bits. In so far as the content of the input is not sufficient to fill the last limb(s) of an element completely, these limbs are supplemented with "0" bits.

There is a special routine developed for the add/sub function of elements. The limbs of the two elements are loaded limb for limb into internal computer registers; the sum and the difference is determined; and the limbs are put back into the elements. This prevents that a copy of an element should be made (as is done in the C^{++} routines), and that subsequently the (modified) copy should be replaced again. In this special routine good records must be kept of any carries or borrows, from the addition as well as from the subtraction.

In the calculation of integers modulo a modulus in the form (2^n+1) the following trick can be used: First divide the number by 2^n. We then get a quotient q and a remainder r. We had to divide by (2^n+1), so we still must reduce the remainder r with q. If r then becomes negative, we must increase the remainder r again with (2^n+1).

In a binary computer, the division of a binary number by (a power of) 2 can happen by means of a shift to the right over one or more bits. The bits which are shifted out (out of the *n* bits of the modulus), form the quotient and the remaining bits constitute the rest, after the quotient is subtracted from the rest.[2].

Another special routine is the Mul/Mod routine. In this routine the multiplication is followed by a mod operation. The value of an element is multiplied by the twiddle factor. Since all the twiddle factors are powers of two, the multiplication may be implemented as a shift to the left over one or more bits (the twiddle-factor represents the number of bits which the content of the element must be shifted to the left). The subsequent operation mod (2^n+1) proceeds as described above.

After the inverse FFT the results should be divided by $2^k \bmod (2^n+1)$. This division by 2^k can be converted into a multiplication with $2^{2n-k} \bmod (2^n+1)$[2]. Multiplying by a power of two can be done by a shift operation to the left.
In negacyclic convolution dividing by 2^k can be combined with dividing by the weighting factor. Converting into a multiplication happens by setting the exponent of two to the value $2n\text{-}k\text{-}w$ where w is the weighting factor.

Testing

Purpose of the test

The aim of the test is to check whether the SSA implementation works well and produces the correct results. Furthermore, to determine the optimum value of *k* at different sizes of input-numbers. Finally, to compare the speed of multiplication between the four methods: The Classical method, the Karatsuba method, the Toom-3 method and the SSA algorithm. Of all these methods specific versions are available for squaring .
A further question is what the best algorithm for the "Dyadic Stage" is. This will be investigated .

The C^{++} implementation is used to check the (partial) results from the assembler implementation and will not be further involved in the testing.

Test Method

For testing, a desktop computer with 4 gigabytes of memory running under Windows 7 is used. A test program is developed that generates a number. This number is input for resp. routines for the Classic method, Karatsuba, Toom-3 and SSA algorithms. The result of the Classic method is stored and the results of the other routines are compared with the Classic result. Furthermore, the number of clock cycles of execution time for each routine is separately measured and reported in the program log.
(This test-program is called: “SSAtest.exe” and can be downloaded from Web-pages www.tonjanee.home.xs4all.nl).
The input number is formed by filling a binary field of predetermined length (number of bits) with at a certain value.
After calculation by the four routines, the input number is increased (in number of bits) by a factor of approximately √2, and this new input number is again processed by the four routines.
This is repeated until the program ends because the requested memory (which is becoming increasingly larger as we work with a greater input number) is no longer available. In this manner, a series of execution times is derived from input numbers of which the size (in numbers of bits) increases logarithmically .
The input numbers can be filled in three different ways and thus form three separate tests, each of which can be activated individually. In the first test, the smallest possible number is formed within the number of significant bits. This is a binary 1 with further only binary zeros. In the second test, the greatest number is formed, which consists of only binary ones . For the third test, the input number is filled with random binary ones and zeros. For this purpose, an LBA routine “PRNG” (Pseudo Random Number Generator) is used. (See below for LBA).
In order to keep the runtimes within reasonable limits with increasing input sizes the routines for resp. Classical, Karatsuba and Toom-3 are switched off. For very large numbers, leaving only the SSA algorithm on. Then no check on the accuracy of the result is possible. (If Classic is off , the result of Karatsuba takes over the role of comparative product. If Karatsuba is disabled, Toom-3 takes over this function). Where during testing an anomaly is

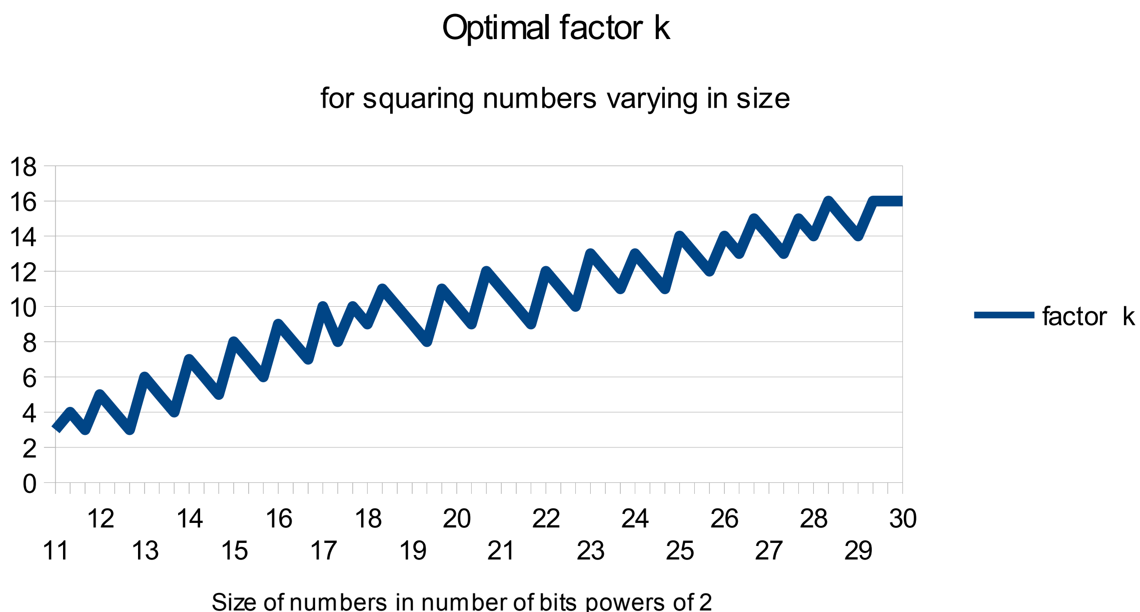
detected in the result of the SSA algorithm, testing is aborted and the error is located and corrected. A program log is kept of each test session.
This test program (with some modifications) is also used to find the optimal value of the factor k for the different input sizes

It is the intention to run this test program on different computers. In order to qualify the recorded number of processor cycles the characteristics of the processor and the number of clock cycles per second of the computer on which the test program is running are recorded in the program log.

Test results

When evaluating the test results, keep in mind the fact that these results are highly dependent on the specific hardware and system software.
First is examined what the best value for *k* is for different sizes of the input for the first level of the SSA algorithm. The lowest value of *k* is 3. In practice, this value will not be used for *k=3* is optimal only for numbers less than approximate 256 bits. These small numbers can be squared much faster with other algorithms.
For numbers bigger than 1.5 Gigabit an optimum value for *k* is 17. However, this large size of input-numbers may let the program stop, depending on the size of main memory of the computer and on the operating system used. In that case the test program stops with a message that there is not enough memory .

The expectation was that the optimal value of *k* would increase proportional to the number of bits input. However, it appears that there is no straight relationship between the size of the input value and *k*, but that this relationship shows more the pattern of a saw tooth.
With increasing input sizes the optimum value of *k* decreases (!) in steps of one to a certain level, and then the optimum value of *k* jumps up three or four steps to a value that is higher than the previously highest value. This pattern repeats itself a number of times.
The following diagram shows these values, which are determined experimentally:



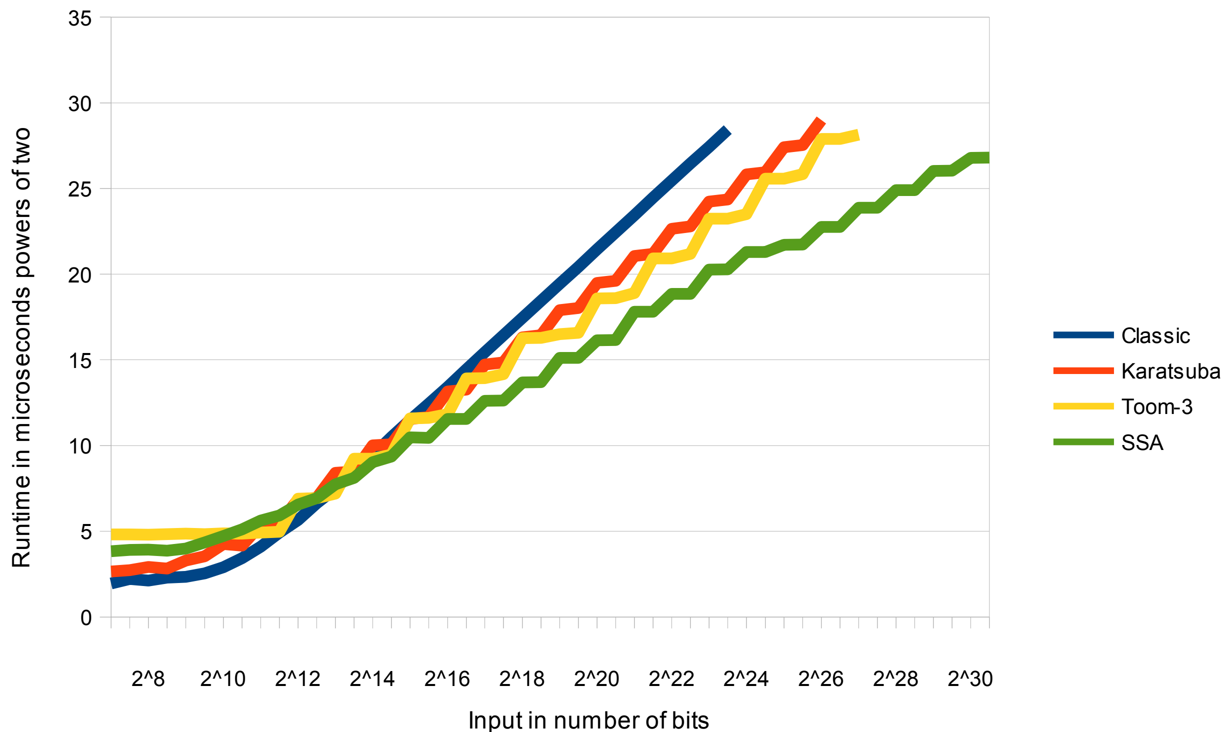
The optimum values for *k* have been inserted into the assembler routine SSA.
Next, we investigated the optimal algorithms for the Dyadic Stage. In the Dyadic Stage elements are squared after the forward butterfly. Elements up to 4,096 bits can best be treated with the Classic algorithm. Elements up to 10,240 bits can best be squared with the Karatsuma algorithm. The Toom3 algorithm is less suitable here because the optimum inputsize in number of bits for Toom3 contains a power of three, while the size of most of the elements consist of only powers of two .
From 10,240 bits the SSA algorithm with the negacyclic convolution can best be applied. Examined is whether the acyclic convolution makes sense to apply. Measurements show

that the acyclic convolution from about 10,240 bits gives similar results as the Karatsuba algorithm, but that the negacyclic convolution is approximately 40 % faster than the acyclic convolution. Although this was to be expected, it had to be tested for completeness.
For the second level Dyadic Stage only the Classic algorithm will be used.
The optimal value of k in the Dyadic Stage is set to 6 for elements of 10,240 bits and up. For elements with size 65,536 bits and up the value for k is set to 7. The use of the negacyclic convolution and these values for k have been inserted in the assembly language implementation.

Subsequently, tests were carried out with ever increasing input-numbers. These numbers are input to squaring operation with resp. routines for the Classical method, the Karatsuba algorithm, the Toom-3 algorithm and the SSA algorithm. Starting with a number of 64 bits, the number of bits were increased each time by a factor of about $\sqrt{2}$. At each multiplication (squaring) the execution time was measured in clock cycles. As much as possible is ensured that the system was "calm" so that there were no other processes along with the active measurement. This could not be entirely prevented. Abnormalities in the measurements are manually corrected.
There are several test runs made. One with as an input a series of binary ones and one with as input a set of random binary ones and zeros. In addition, a test run with as input one binary 1 and further only binary zeros . This is actually a nonsensical run because the input is a pure power of two. Squaring is a matter of doubling the exponent of two. Nevertheless, this has proved to be a valuable test to detect errors.
The analysis of the results shows that the required number of computer cycles for squaring, as expected, increases as the input, expressed in number of significant bits, increases. The extent to which that number of cycles was growing is highly dependent on the algorithm that is being used. But the analysis also shows that the number of cycles required depends on the type of input .
A control-run on a different computer than the test computer also shows that the variance among the four algorithms, is also influenced by the processor-type and the operating system used. Therefore, the conclusions of the measurements should be used with caution.

The measurement of the testrun with **random** input we consider as the base case, because in practice input-numbers may be considered also as random. The test with the smallest value as input (ie, one binary 1 and further only binary zeros) requires, compared to the base case, 25 % less cycles for the Classic method; approximately 15 % less cycles for Karatsuba and Toom-3, and about 30 % less cycles for SSA algorithm. The test with the highest value as input (ie all binary 1) requires compared to the base case more cycles; on average 40 % more for Classic; an average of 4 % more for Karatsuba; for Toom-3, an average of 4 % more cycles for the smaller numbers, but about 3 % fewer cycles for the larger numbers . For SSA, the picture is mixed, but for larger numbers, there is hardly any difference in the number of cycles compared to the base case.

The mutual comparison between the algorithms is hampered by the differences in test results between the test runs with different input. Also, test runs on other computers do not provide more clarity. Therefore it was decided to further only proceed with the results of the base case (random input):



The test results are included in the above depicted graph. (The designation 2^7 .... etc. underneath the graph represents 2^7 .. etc, and gives the size of the inputnumber, expressed in number of bits.) The results of the other tests confirm the trend shown in the graph, but are not further analysed.

In the graph, the processing time is displayed for the squaring of random numbers with logarithmically increasing numbers of bits input for the four algorithms. The processing time (or runtime) is expressed in number of microseconds; those numbers are shown as powers of two (also logarithmic). The horizontal line at 20 indicates 2^{20} microseconds, which is approximately one second. The time is calculated by dividing the number of computer cycles by the number of cycles per second of the relevant processor.

We see here that the classic method gives the fastest results to about 4,000 bits . From about 11,000 bits the SSA algorithm is the fastest (that applies to all three types of input). Between 4,000 and 11,000 bits the four algorithms do not differ much when it comes to speed; with increasing numbers of bits between 4 and 11 thousand the four algorithms interchange when it comes to what is the fastest .

Conclusions

The speed (in number of computer cycles) with which a number can be squared is not only dependent on the size of that number, and the algorithm used, but also on the composition of that number, and the computer hardware and software used . Therefore, the following statements are not generally valid, but they are only valid for the algorithms as implemented here, and the computing environment in which these algorithms are tested.

Squaring smaller numbers (numbers less than about 1,200 decimal digits) is still the fastest with the classical algorithm. Numbers from approximately 3,300 decimal digits can be squared the fastest with the SSA algorithm. Between about 1,200 and 3,300 decimal digits the four algorithms do not differ much when it comes to speed; with increasing numbers of bits input the four algorithms interchange when it comes to what is the fastest .

Calculating the square of a number with one million decimal digits can be done on any modern computer running a 32-bit Windows operating system with the SSA implementation described here within only one second. The largest number that can be squared with this implementation on a computer with 4 gigabytes of memory, running as a 32-bit task under Windows 7, is a number of 1.5 Gigabit or almost 500 million decimal digits. The runtime is about 2 minutes .

The SSA algorithm (with negacyclic convolution) is actually meant to be used recursively. With the described SSA implementation this is not really the case. For the first level of implementation, the acyclic convolution is used because it is simpler than the negacyclic convolution and gives the full square .
On the second level (in the Dyadic Stage) for numbers to about 10 million decimal digits input the Classical or Karatsuba algorithm is used. For numbers greater than about 10 million decimal digits input, the SSA algorithm with the negacyclic convolution, is used in the Dyadic Stage. The negacyclic convolution is implemented as a separate routine, which is different from the routine for the acyclic convolution for the first level. The acyclic convolution on the second level offers no advantages and uses approximately 40 % more computer cycles than the negacyclic convoution.
At the third level, the Dyadic Stage only uses the Classic algorithm. So there is no recursion .

The use of the recursive negacyclic convolution for numbers that are greater than about 500 million decimal digits may be an option. However, this is beyond the capacity of the used test environment (hardware and software). Also the investigation of calculations with numbers over 500 million decimal digits is beyond the scope of this document.

There are many further improvements possible in the implementation of the SSA algorithm described. A transition to a 64-bit architecture will undoubtedly provide substantial acceleration. In addition, in the literature, many suggestions and comments that may or may not be successfully implemented in SSA products[3] can be found.

Example program

The following is an example program in C^{++} code, in which the SSA algorithm is implemented. Both the acyclic convolution and the negacyclic convolution are included . Only the relevant code is shown. It is assumed that the container is defined with enough elements and that the elements are of sufficient size. A square is calculated, so that there is only one input number, and therefore only one container. The X in the program is a table of address pointers to the elements of the container.

For both convolutions, k is set to 5 in the program, so the number of elements of the container is $2^5 = 32$. The size of the elements is set to 64 bits, the modulus used is $2^{64}+1$. (In reality, the elements are greater than 64 bits, so that the value 2^{64} can be stored).
The input number consist of a specified number of bits that will be filled with random data. For this, a special routine PRNG (Pseudo Random Number Generator) is used.

With the acyclic convolution only half of the elements is filled at the start with input-data. The other half of elements is set to zero (zero-padding). The "start-content" (this is the maximum number of bits with which an element may be filled at the beginning of the FFT process) is calculated different than as described above, and is $(64/2)\text{-}k = 32 - k = 27$ bits (rounded). This prevents the necessity of applying the modulus (2^n+1) to the division by N. The input is a random number with 432 bits, so that all 16 elements can be filled (16 x 27 bits) completely .

In the negacyclic convolution all elements are used from the start of the FFT process. The"start-content" is calculated as described above, and is (size-of-element - k) / 2 = $(64 - k) / 2 = 59/2 = 29$ bits (rounded). The input consists of a random number of 928 bits by which the 32 elements are completely filled (32x29 bits).

In the program log the input number is shown, as well as the square, calculated via the Classical method with the function “Smul”. Then, the square is calculated according to the SSA algorithm and is displayed in the program log for comparison with the results of the Classical method.
The calculations for the acyclic and for the negacyclic convolution can be activated separately in the SSA test program and are shown in the sample code in succession .

The example program in C^{++} code uses a number of functions that is required for calculations with large numbers. These functions belong to the LBA architecture and are described later in this document .

Example C^{++} Code

```
//==========================================================================
//        SSA Algorithm - Acyclic convolution
//--------------------------------------------------------------------------
//        Definition of SSA Integers
//--------------------------------------------------------------------------
          int     ReqSize        ; // required number of bits in input
          int     Sel            ; // size of one element in # bits
          int     k              ; // exponent for two for
          int     D              ; // # elements in container
          int     Content        ; // start-content of one element # bits
//--------------------------------------------------------------------------
//        Definition of FFT variables
//--------------------------------------------------------------------------
          void*   p              ; // address-pointers to elements
          void*   q              ; // of container
          int     a              ; // twiddle factor (exponent of 2)
          int     g              ; // root of unity (exponent of 2)
          int     i              ; // i,j and m are indices
          int     j              ; // used in Gentleman Sande and
          int     m              ; // Cooley-Tukey loops
          int     n              ; // number of elements in container
          int     t              ; // exponent for conversion div in mul
//--------------------------------------------------------------------------
//        Preparation of input
//--------------------------------------------------------------------------
          Mes (31,0)             ; // write separation line in log
          Mes (53,0)             ; // write "Action 6" in log
          Mes (59,0)             ; // write "Test SSA C++ acyclic" in log
          ReqSize = 464          ; // requested size of input in bits
          Prng (input)           ; // fill input with random bits
          Nosb (n,input)         ; // number of significant bits in n
          while (n>ReqSize)   {    // as long as n greater than ReqSize
             Shr (input,1)       ; // divide input by two
             Nosb (n,input) ; }  ; // and calculate Nosb again
//--------------------------------------------------------------------------
//        Preparation FFT variables (Container)
//--------------------------------------------------------------------------
          k = 5                  ; // exp. of 2 to define # elements
          Sel = 64               ; // size of one element in # bits
          D = 1                  ; // calculate D (# of elements
          for (i=0; i<k; i=i+1)    // in container)
              D = D*2 ;            // D = 2^k
          g = 2*Sel/D            ; // calculate # bits for root of unity
          Load (modulus,0)       ; // set modulus to 0
          Bts  (modulus,Sel)     ; // calculate:
          Inc  (modulus)         ; // modulus = (2^Sel) + 1
          Content = (Sel-k)/2    ; // calculate start-content
          Load (ContLba,0)       ; // calculate: multiplier in
          Bts (ContLba,Content)  ; // Lba field for Content
//--------------------------------------------------------------------------
//        Report SSA & FFT variables
//--------------------------------------------------------------------------
          Mes    (31,0)          ; // write separation line via message
          Nosb   (n,input)       ; // determine number of significant
          Mes    (64,n)          ; // bits in input and report
          Mes    (65,k)          ; // report k
          Mes    (66,D)          ; // report # elements in container
          Mes    (69,Sel)        ; // report size of element
          Mes    (70,g)          ; // report root of unity
          Mes    (71,Content)    ; // start-content # bits per element
```

```
          Mes    (31,0)          ; // write separation line via message
          Report (modulus)       ; // report modulus
          Mes    (31,0)          ; // write separation line via message
          Report (ContLba)       ; // report start-content in Lba form
          Mes    (31,0)          ; // write separation line via message
//-------------------------------------------------------------------------
//        Calculate square-product in classic way and report
//-------------------------------------------------------------------------
          Report (input)                  ; // report value of input
          Mes    (31,0)                   ; // write separation line
          Smul   (RC,SquareClassic,input); // square multiply input
          Report (SquareClassic)          ; // and report result
          Mes    (31,0)                   ; // write separation line
//-------------------------------------------------------------------------
//        Load elements of container with input
//-------------------------------------------------------------------------
          for (i=0;i<D/2;i=i+1) { // for first half elements of
            p = X[i]            ; // container load address from
            Mov (wrk,input)     ; // address-array X in p; copy input to
            Mod (wrk,ContLba)   ; // workfield, extract content # bits
            Mov (p,wrk)         ; // and copy extract to element p;
            Shr (input,Content) ; // divide input by shifting bits right
               }                ; // Zero-padding:
          for (i=D/2;i<D;i=i+1) { // for second half elements of
            p = X[i]            ; // container load address from
            Load (p,0)          ; // address-array X in p; load zero
               }                ; // in element p
//-------------------------------------------------------------------------
//        Forward FFT - Gentleman-Sande
//        See "Prime Numbers" 2d Edition p.480, ISBN-13 978-0387-25282-7
//-------------------------------------------------------------------------
          n = D ;                                 // n # elements container
          for ( m = n/2 ; m >= 1 ; m = m/2 )      // start with m = half n
              {                                   // dividing m by two until
              for ( j = 0; j < m; j = j + 1 )     // m = 1; start with j =0
                  {                               // increasing j by 1 until
                  a = (g*j*n) / (2*m)         ; // j>=m; calculate twiddle
                  for (i=j; i < n; i=i+2*m )      // factor; place indices
                      {                           // to elements in
                         p = X[i] ;               // container in p
                         q = X[i+m] ;             // and q
//
//        AddSub
//
          Mov (wrk,p)             ; // copy element p in workfield
          Add (p,q)               ; // add element q to element p
          Mod (p,modulus)         ; // apply modulus on p
          while (LessThan (wrk,q))  // if workfield (original p) less
                Add (wrk,modulus) ; // than q, add modulus to workfield
          Sub (wrk,q)             ; // subtract q from workfield
          Mov (q,wrk)             ; // copy workfield in q
//
//        MulMod
//
          if (a != 0)           {   // do only if Twiddle-Factor not
            Shl  (q,a)            ; // is zero: multiply q by Twiddle
            Mod  (q,modulus)      ; // Factor (by shifting left) and
                                } ; // apply modulus on q
                        } ; } ;
                                } ; // end of foreward FFT
//        End of Gentleman-Sande FFT
```

```
//----------------------------------------------------------------------------
//        Dyadic stage - Square multiplication of elements
//----------------------------------------------------------------------------
          for (i=0; i<D; i=i+1)   // for every element
          {                       // in container load address of
            p = X[i]         ;    // element from address-array X in p
            Mov  (wrk,p)     ;    // calculate square of
            Smul (RC,p,wrk)  ;    // element p in p
            Mod  (p,modulus) ;    // and apply modulus
          }                  ;    // on element p
//----------------------------------------------------------------------------
//        Inverse FFT - Cooley-Tukey
//        See "Prime Numbers" 2d Edition p.480, ISBN-13 978-0387-25282-7
//----------------------------------------------------------------------------
          for ( m = 1 ; m < n ; m = 2*m )     // start with m = 1,multi-
              {                               // plying m by two until
              for ( j=0; j < m; j=j+1)        // m not < n; strt with j=0
                  {                           // increasing j by 1 until
                  a = (g*j*n) / (2*m) ;       // j not < m; calculate
                  for (i=j; i < n; i=i+2*m)   // twiddle factor;
                      {                       // place addresses to
                         p = X[i] ;           // elements in container
                         q = X[i+m] ;         // in p and q
//
//        MulMod
//
          if (a != 0)             // do only if Twiddle Factor not is
              {                   // zero: multiply q by Twiddle Factor
              Shl  (q,a)        ; // (by shifting left) and apply
              Mod  (q,modulus)  ; // modulus on element q
              }                 ; //
//
//        AddSub
//
          while (LessThan (p,q))  // if p < q, then
                Add (p,modulus) ; // add modulus to p
          Mov (wrk,p) ;           // copy p to workfield
          Add (p,q)   ;           // add q to p
          Mod (p,modulus) ;       // apply modulus to p
          Sub (wrk,q) ;           // subtract q from workfield
          Mov (q,wrk) ;           // copy workfield to q
            } ; } ; } ;           // end of backward FFT
//
//        End of Cooley-Tukey FFT
//----------------------------------------------------------------------------
//        Divide every element by 2^k (mod modulus)
//----------------------------------------------------------------------------
          t = 2*Sel - k         ; // convert division into multiplication
          for (i=0; i<D; i=i+1) { // multiply every element by 2^t
            p = X [i]           ; // by shifting t bits
            Shl (p,t)           ; // to the left
            Mod (p,modulus) ; } ; // apply modulus
          X[D] = X[0]           ; // place index first element
                                  // as last in table
//----------------------------------------------------------------------------
//        Composition of final result and report
//-------------------------------------------------------------------------
          Load (SquareSSA,0)         ; // init SquareSSA to zero
          for (i=1; i<=D ; i=i+1)      // process all elements of
          {                            // container, from first to last
             p = X[i]                ; // load address to element in p
```

```
            Shl (SquareSSA,Content) ; // multiply p by 2^Content (by
            Add (SquareSSA,p)       ; // shifting content bits left)
         }                          ; // add element p to result
         Report (SquareSSA)         ; // report result
         Mes (31,0)                 ; // write separation line
//----------------------------------------------------------------------
//       End of SSA algorithm - acyclic convolution
//======================================================================
```

```
//=========================================================================
//        SSA Algorithm - negacyclic convolution
//-------------------------------------------------------------------------
//        Definition of SSA Integers
//-------------------------------------------------------------------------
          int     ReqSize       ; // required number of bits in input
          int     Sel           ; // size of one element in # bits
          int     k             ; // exponent for two for
          int     D             ; // # elements in container
          int     Content       ; // start-content of one element # bits
//-------------------------------------------------------------------------
//        Definition of FFT and DWT variables
//-------------------------------------------------------------------------
          void*   p             ; // address-pointers to elements
          void*   q             ; // of container
          int     a             ; // twiddle factor (exponent of 2)
          int     g             ; // root of unity (exponent of 2)
          int     i             ; // i,j and m are indices
          int     j             ; // used in Gentleman Sande and
          int     m             ; // Cooley-Tukey loops
          int     n             ; // number of elements in container
          int     t             ; // exponent for conversion div in mul
          int     Th            ; // (Theta) general weight factor
          int     Wf            ; // weight factor for element
          int     Sn            ; // workfield for sequnce number element
//-------------------------------------------------------------------------
//        Preparation of input
//-------------------------------------------------------------------------
          Mes (31,0)            ; // write separation line in log
          Mes (62,0)            ; // write "Action 7" in log; write
          Mes (82,0)            ; // "Test SSA C++ negacyclic" in log
          ReqSize = 928         ; // requested size of input in bits
          Prng (input)          ; // fill input with random bits
          Nosb (n,input)        ; // number of significant bits in n
          while (n>ReqSize)   {   // as long as n greater than ReqSize
             Shr (input,1)      ; // divide input by two
             Nosb (n,input) ; } ; // and calculate Nosb again
//-------------------------------------------------------------------------
//        Preparation FFT variables (Container)
//-------------------------------------------------------------------------
          k = 5                 ; // exp. of 2 to define # elements
          Sel = 64              ; // size of one element in # bits
          D = 1                 ; // calculate D (# of elements
          for (i=0; i<k; i=i+1)   // in container)
              D = D*2 ;           // D = 2^k
          g = 2*Sel/D           ; // calculate # bits for root of unity
          Content = (Sel-k)/2   ; // calculate start-content
          Load (ContLba,0)      ; // calculate: multiplier in
          Bts (ContLba,Content) ; // Lba field for Content
          n = Content*D         ; // calculate capacity container in bits
          Load (modulusN,0)     ; // set modulusN to 0
          Bts  (modulusN,n)     ; // calculate modulusN :
          Inc  (modulusN)       ; // modulusN = (2^n + 1)
          Load (modulus,0)      ; // set modulus for element to 0
          Bts  (modulus,Sel)    ; // calculate modulus for element
          Inc  (modulus)        ; // modulus = (2^Sel + 1)
//-------------------------------------------------------------------------
//        Report SSA & FFT variables
//-------------------------------------------------------------------------
          Mes    (31,0)         ; // write separation line via message
          Nosb   (n,input)      ; // determine number of significant
```

```
          Mes    (64,n)          ; // bits in input and report
          Mes    (65,k)          ; // report k
          Mes    (66,D)          ; // report # elements in container
          Mes    (69,Sel)        ; // report size of element
          Mes    (70,g)          ; // report root of unity
          Mes    (71,Content)    ; // start-content # bits per element
          Mes    (31,0)          ; // write separation line
          Report (modulus)       ; // report modulus
          Mes    (31,0)          ; // write separation line
          Report (modulusN)      ; // report modulusN
          Mes    (31,0)          ; // write separation line
          Report (ContLba)       ; // report start-content of element
          Mes    (31,0)          ; // in Lba form; write separation line
//---------------------------------------------------------------------------
//        Calculate square-product in classic way and report
//---------------------------------------------------------------------------
          Report (input)                  ; // report value of input
          Mes    (31,0)                   ; // write separation line
          Smul   (RC,SquareClassic,input); // square multiply input
          Report (SquareClassic)          ; // and report result
          Mes    (31,0)                   ; // write separation line
//---------------------------------------------------------------------------
//        Load elements of container with input
//---------------------------------------------------------------------------
          for (i=0;i<D;i=i+1) {    // for all elements of
            p = X[i]             ; // container load address from
            Mov (wrk,input)      ; // address-array X in p; copy input to
            Mod (wrk,ContLba)    ; // workfield, extract content # bits
            Mov (p,wrk)          ; // and copy extract to element p;
            Shr (input,Content) ; // divide input by shifting bits right
               }                 ; //
//---------------------------------------------------------------------------
//        Apply weight factor
//---------------------------------------------------------------------------
          Th = Sel/D           ; // calculate general weight factor
          for (i=0;i<D;i=i+1) { // for all elements of container
             Wf = i*Th         ; // calculate weight factor for element
             if (Wf != 0)   {    // do only if weight Factor not zero
             p = X[i]          ; // load address from address array
             Shl  (p,Wf)       ; // multiply p by Weight factor
             Mod  (p,modulus) ; // (by shifting left) and
                      } ; } ; // apply modulus on p
//---------------------------------------------------------------------------
//        Forward FFT - Gentleman-Sande
//        See "Prime Numbers" 2d Edition p.480, ISBN-13 978-0387-25282-7
//---------------------------------------------------------------------------
          n = D ;                               // n # elements container
          for ( m = n/2 ; m >= 1 ; m = m/2 )    // start with m = half n
              {                                 // dividing m by two until
              for ( j = 0; j < m; j = j + 1 )   // m = 1; start with j =0
                  {                             // increasing j by 1 until
                  a = (g*j*n) / (2*m)         ; // j>=m; calculate twiddle
                  for (i=j; i < n; i=i+2*m )    // factor; place indices
                      {                         // to elements in
                         p = X[i] ;             // container in p
                         q = X[i+m] ;           // and q
//
//        AddSub
//
          Mov (wrk,p)              ; // copy element p in workfield
          Add (p,q)                ; // add element q to element p
```

```
          Mod (p,modulus)         ; // apply modulus on p
          while (LessThan (wrk,q))  // if workfield (original p) less
                Add (wrk,modulus) ; // than q, add modulus to workfield
          Sub (wrk,q)             ; // subtract q from workfield
          Mov (q,wrk)             ; // copy workfield in q
//
//        MulMod
//
          if (a != 0)           {   // do only if Twiddle-Factor not
            Shl  (q,a)            ; // is zero: multiply q by Twiddle
            Mod  (q,modulus)      ; // factor (by shifting left) and
                                } ; // apply modulus on q
                        } ; } ; } ; // end of forward FFT

//        End of Gentleman-Sande FFT
//----------------------------------------------------------------------
//        Dyadic Stage - Square multiplication of elements
//----------------------------------------------------------------------
          for (i=0; i<D; i=i+1)   // for every element
          {                       // in container load address of
            p = X[i]         ;    // element from address-array X in p
            Mov  (wrk,p)     ;    // calculate square of
            Smul (RC,p,wrk)  ;    // element p in p
            Mod  (p,modulus) ;    // and apply modulus
          }                  ;    // on element p
//----------------------------------------------------------------------
//        Inverse FFT - Cooley-Tukey
//        See "Prime Numbers" 2d Edition p.480, ISBN-13 978-0387-25282-7
//----------------------------------------------------------------------
          for ( m = 1 ; m < n ; m = 2*m )      // start with m = 1,multi-
              {                                // plying m by two until
              for ( j=0; j < m; j=j+1)         // m not < n; strt with j=0
                  {                            // increasing j by 1 until
                  a = (g*j*n) / (2*m) ;        // j not < m; calculate
                  for (i=j; i < n; i=i+2*m)    // twiddle factor;
                      {                        // place addresses to
                         p = X[i] ;            // elements in container
                         q = X[i+m] ;          // in p and q
//
//        MulMod
//
          if (a != 0)             // do only if Twiddle Factor not is
              {                   // zero: multiply q by Twiddle Factor
              Shl  (q,a)        ; // (by shifting left) and apply
              Mod  (q,modulus)  ; // modulus on element q
              }                 ; //
//
//        AddSub
//
          while (LessThan (p,q))  // if p < q, then
                Add (p,modulus) ; // add modulus to p
          Mov (wrk,p) ;           // copy p to workfield
          Add (p,q)   ;           // add q to p
          Mod (p,modulus) ;       // apply modulus to p
          Sub (wrk,q) ;           // subtract q from workfield
          Mov (q,wrk) ;           // copy workfield to q
            } ; } ; } ;           // end of backward FFT
//
//        End of Cooley-Tukey FFT
```

```
//-----------------------------------------------------------------------
//         Apply inverse weight factor & divide elements by k(mod modulus)
//-----------------------------------------------------------------------
          X[D] = X[0]          ; // place index first element as last in
                                 // in address table; for every element
          for (i=1;i<=D;i=i+1)  { // of container from 1 to D convert
            t = 2*Sel-(D-i)*Th-k; // division in multiplication by calcu-
            p = X [i]          ; // lating number of bits to shift
            Shl (p,t)          ; // to the left
            Mod (p,modulus) ; } ; // apply modulus
//-----------------------------------------------------------------------
//         Composition of final result
//-----------------------------------------------------------------------
          t = 2*Content              ; // prepare integer with 2*Content
          Mov (SquareSSA,modulusN)   ; // init SSA result with modulusN
          for (i=1; i<=D ; i=i+1)      // process all elements of
          {                            // container, from first to last
            p = X[i]                 ; // load address to element in p
            Shl (SquareSSA,Content)  ; // multiply result by 2^Content
            Add (SquareSSA,p)        ; // add p to result
            Sn=(D+1-i)               ; // calculate sequence number of
            Load (wrk2,Sn)           ; // element; load number in wrk
            Shl (wrk2,t)             ; // by shifting t bits left
            if (LessThan (wrk2,p)) {   // if p >= wrk2, then correction:
              Sub (SquareSSA,modulus); // subtract modulus
                               } ; } ; // from result
//-----------------------------------------------------------------------
          Report (SquareSSA)         ; // report Square SSA without
          Mes (31,0)                 ; // modulusN
//--------------------------------------------------------------------
//         Calculate final SSA product mod modulusN and report
//--------------------------------------------------------------------
          Mov (SquareSSA_mod_modulusN,SquareSSA) ; // copy SquareSSA
          Mod (SquareSSA_mod_modulusN,modulusN)  ; // apply modulusN
          Report (SquareSSA_mod_modulusN)        ; // report SquareSSA
          Mes    (31,0)                          ; // write separation
//--------------------------------------------------------------------
//         End of SSA algorithm - negacyclic convolution
//--------------------------------------------------------------------
//         To check result of SSA product;
//         calculate Classic product mod modulusN and report
//--------------------------------------------------------------------
          Mov (SquareClassic_mod_modulusN,SquareClassic) ; // apply
          Mod (SquareClassic_mod_modulusN,modulusN)  ; // modulusN on
          Report (SquareClassic_mod_modulusN)        ; // classic
          Mes    (31,0)                              ; // write separ.
//====================================================================
```

Program Log

```
14:49:42 Storage logging enabeled
14:49:42 Disk logging enabeled
14:49:42 Set language to English
14:49:42 --------------------------------------------------------------------
14:49:42 Processor information:
14:49:42    Vendor ID =   GenuineIntel
14:49:42    Processor Type = Original processor
14:49:42    Processor Family =  06H
14:49:42    Processor Model  =  25H
14:49:42    Stepping ID =   02H
14:49:44 Number_of_cycles_per_second =                        3,192,035,784
14:49:44 --------------------------------------------------------------------
14:49:44 SSA Test Program Version 10.0
14:49:44 Date                          02/25/2014
14:49:44 --------------------------------------------------------------------
14:49:52 --------------------------------------------------------------------
14:49:52 Action 6:
14:49:52 Test SSA C++ acyclic convolution
14:49:52 --------------------------------------------------------------------
14:49:52 Input (# bits)                                                    464
14:49:52 k =                                                                 5
14:49:52 D (number of elements in container) =                              32
14:49:52 Element size (# bits) =                                            64
14:49:52 root of unity in exponent for 2 (g) =                               4
14:49:52 Start-content (# bits) per element =                               29
14:49:52 --------------------------------------------------------------------
14:49:52 modulus =                                  18,446,744,073,709,551,617
14:49:52 --------------------------------------------------------------------
14:49:52 ContLba =                                                 536,870,912
14:49:52 --------------------------------------------------------------------
14:49:52 input =                   23,880,974,640,406,874,006,561,095,314,759
    ,812,195,087,890,202,713,396,697,036,617,948,702,507,249,760,510,653,564
    ,284,562,865,444,417,319,960,656,828,998,673,925,084,333,072,643,081,903
14:49:52 --------------------------------------------------------------------
14:49:52 SquareClassic =                                           570,300,949
    ,775,756,225,264,886,968,102,018,863,677,356,416,387,697,710,959,059,828
    ,930,229,979,283,007,679,759,491,039,249,378,781,325,900,381,477,624,283
    ,009,704,564,797,346,174,409,871,148,652,489,961,706,551,129,278,520,198
    ,255,868,379,802,679,790,699,161,145,794,347,195,544,420,062,737,001,342
    ,741,805,289,222,297,494,495,032,891,472,381,879,268,746,365,966,101,409
14:49:52 --------------------------------------------------------------------
14:49:52 SquareSSA =                                               570,300,949
    ,775,756,225,264,886,968,102,018,863,677,356,416,387,697,710,959,059,828
    ,930,229,979,283,007,679,759,491,039,249,378,781,325,900,381,477,624,283
    ,009,704,564,797,346,174,409,871,148,652,489,961,706,551,129,278,520,198
    ,255,868,379,802,679,790,699,161,145,794,347,195,544,420,062,737,001,342
    ,741,805,289,222,297,494,495,032,891,472,381,879,268,746,365,966,101,409
14:49:52 --------------------------------------------------------------------
14:49:59 --------------------------------------------------------------------
14:49:59 Action 7:
14:49:59 Test SSA C++ negacyclic convolution
14:49:59 --------------------------------------------------------------------
14:49:59 Input (# bits)                                                    928
14:49:59 k =                                                                 5
14:49:59 D (number of elements in container) =                              32
14:49:59 Element size (# bits) =                                            64
14:49:59 root of unity in exponent for 2 (g) =                               4
14:49:59 Start-content (# bits) per element =                               29
```

```
14:49:59 -----------------------------------------------------------------
14:49:59 modulus =                                    18,446,744,073,709,551,617
14:49:59 -----------------------------------------------------------------
14:49:59 modulusN =                                                2,269,007,733
    ,883,335,972,287,082,669,296,112,915,239,349,672,942,191,252,221,331,572
    ,442,536,403,137,824,056,312,817,862,695,551,072,066,953,619,064,625,508
    ,194,663,368,599,769,448,406,663,254,670,871,573,830,845,597,595,897,613
    ,333,042,429,214,224,697,474,472,410,882,236,254,024,057,110,212,260,250
    ,671,521,235,807,709,272,244,389,361,641,091,086,035,023,229,622,419,457
14:49:59 -----------------------------------------------------------------
14:49:59 ContLba =                                                   536,870,912
14:49:59 -----------------------------------------------------------------
14:49:59 input =                                                   1,912,854,700
    ,240,703,247,807,367,776,187,277,693,136,436,476,403,764,002,672,977,962
    ,872,747,530,537,541,320,974,025,779,848,557,429,328,414,586,156,528,095
    ,960,627,452,388,892,414,573,230,890,551,763,151,497,582,351,925,632,448
    ,987,016,277,532,769,096,937,874,072,180,705,501,084,149,648,213,673,329
    ,751,513,948,338,582,722,534,765,613,680,676,492,180,381,567,620,394,449
14:49:59 -----------------------------------------------------------------
14:49:59 SquareClassic =                            3,659,013,104,232,950,677
    ,805,114,344,121,779,531,197,117,277,963,568,013,405,909,974,870,598,288
    ,886,477,617,692,701,918,135,397,168,605,437,099,177,155,532,495,620,439
    ,448,843,639,383,753,343,099,263,432,987,962,908,415,752,284,348,508,852
    ,220,704,071,640,371,102,730,979,856,850,949,166,463,205,942,476,246,997
    ,038,089,521,576,070,314,504,185,845,955,884,796,709,432,927,069,966,851
    ,912,347,541,033,315,963,647,546,540,875,319,099,425,769,661,434,963,508
    ,496,692,230,688,148,550,225,759,061,844,579,466,295,441,731,527,700,259
    ,787,871,999,218,741,972,932,470,720,043,938,617,922,192,781,180,349,790
    ,894,528,609,053,503,636,801,152,877,585,335,957,504,905,877,102,525,782
    ,352,013,132,684,051,783,013,723,746,340,376,751,122,332,438,350,013,601
14:49:59 -----------------------------------------------------------------
14:49:59 SquareSSA =                                5,148,396,096,422,391,593
    ,693,210,985,222,689,424,575,005,282,213,296,071,655,104,164,476,572,739
    ,631,099,228,161,281,473,378,140,200,867,014,462,091,313,754,559,770,129
    ,599,777,987,619,078,360,817,109,726,631,408,021,465,198,022,633,853,290
    ,201,484,467,517,254,546,983,876,930,815,720,308,649,330,475,055,891,564
    ,430,659,129,037,097,735,228,607,981,843,261,103,881,339,986,776,030,856
    ,576,383,602,650,196,863,786,194,589,864,861,073,949,213,537,908,662,857
    ,442,346,123,139,973,727,401,088,620,534,702,813,367,631,813,402,895,345
    ,245,666,594,639,646,138,936,032,404,782,138,327,221,696,603,828,925,003
    ,013,479,874,303,961,645,996,162,695,943,063,420,105,368,768,690,469,502
    ,641,987,635,733,388,334,344,364,203,665,897,629,938,805,243,073,923,390
14:49:59 -----------------------------------------------------------------
14:49:59 SquareSSA_mod_modulusN =                                  1,097,245,176
    ,442,419,787,316,231,653,788,707,639,793,027,550,241,263,859,044,448,769
    ,583,195,800,977,169,253,071,816,654,060,500,772,836,228,452,142,675,638
    ,745,326,844,877,097,311,691,459,597,422,276,292,407,497,203,346,140,587
    ,936,617,058,395,410,214,925,715,151,939,903,044,305,437,420,080,796,980
    ,738,622,252,001,243,848,252,936,694,677,332,920,144,298,245,544,292,274
14:49:59 -----------------------------------------------------------------
14:49:59 SquareClassic_mod_modulusN =                              1,097,245,176
    ,442,419,787,316,231,653,788,707,639,793,027,550,241,263,859,044,448,769
    ,583,195,800,977,169,253,071,816,654,060,500,772,836,228,452,142,675,638
    ,745,326,844,877,097,311,691,459,597,422,276,292,407,497,203,346,140,587
    ,936,617,058,395,410,214,925,715,151,939,903,044,305,437,420,080,796,980
    ,738,622,252,001,243,848,252,936,694,677,332,920,144,298,245,544,292,274
14:49:59 -----------------------------------------------------------------
14:50:04 Logging disabled
```

Lba (Long Binary Architecture)

LBA is a (software) architecture to perform calculations with large numbers. In the computer world, large numbers are usually called “Bignums” and calculations with big numbers are called Arbitrary Precision Arithmetic.
LBA (Long Binary Arithmetic) is a form of Arbitrary Precision Arithmetic and works only with natural numbers, ie whole positive numbers and zero . There is a function available for subtracting two LBA variables from each other, but if the difference is negative then the result is unpredictable.
LBA describes the format of variables and provides a library of functions that can be applied to these variables . These functions can be called from assembler programs and C^{++} programs.
LBA is available for a 32 -bit Windows operating environment on Intel x86 architecture.
For a large number of arithmetic and logic functions, for which a computer instruction is available in the Intel x86 architecture, there is also a function present in the LBA architecture. Examples are :
Add, Sub, Mul , Div , Inc , Dec, Bts , Bsf .

In the example of the SSA algorithm in C^{++} code the following LBA functions are used:

Function	Parameters	Description
Mov	(p1,p2)	Copy the content of p2 to p1
Load	(p1,int)	Put 0 in p1 and copy int into 1^e limb of p1
Add	(p1,p2)	Add p2 to p1
Sub	(p1,p2)	Subtract p2 from p1
Mod	(p1,p2)	Divide p1 by p2 and place remainder in p1
Shr	(p1,int)	Shift p1 right the number of bits in int
Shl	(p1,int)	Shift p1 left the number of bits in int
Bts	(p1,int)	Put a “1” bit in p1 on bit position number in int
Inc	(p1)	Increase p1 by one
Dec	(p1)	Decrease p1 by one
Nosb	(int,p1)	The number of significant bits of p1 is put in int
LessThan	(p1,p2)	if p1 < p2 then execute the following instructions
Smul	(RC,p1,p2)	Calculate the square of p2 and put the result in p1
		This calculation is based on the classical method
Report	(p1)	Report Lba-variable p1 in the Program Log
Mes	(n,int)	Write message number n to the Program Log, comple-
		ted with the value of the 32-bits integer from int
Prng	(p1)	Pseudo Random Number Generator; p1 is filled with
		random binary ones and zero's.

p1 and p2 are pointers to Lba variables.
int is a 32 bits integer (dword).
RC is Result Code; RC=0 is the normal situation

References

1. Richard Crandall & Carl Pomerance. *Prime Numbers – A Computational Perspective*. Second Edition, Springer, 2005. Chapter 9: Fast Algorithms for large-integer arithmetic, pages 480, 502, 507.
2. Schönhage-Strassen algorithm (http://en.wikipedia.org/wiki/Schönhage_Strassen_Algorithm) pages 2 & 4.
3. Pierrick Gaudry, Alexander Kruppa, Paul Zimmerman. *A GMP-based Implementation of Schönhage-Strassen's Large Integer Multiplication Algorithm* (http://www.loria.fr/~gaudry/publis/issac07.pdf). Proceedings of the 2007 International Symposium on Symbolic and Algebraic Computation.
4. Sergey Chernenko. *Fast Fourier transform – FFT* (http://www.librow.com/articles/article-10).
5. *Number-theoretic transform. (*http://en.wikipedia.org/wiki/Number-theoretic_transform)
6. *Big O notation. (http://en.wikipedia.org/wiki/Big_O_notation)*
7. *Root of unity modulo n. (http://en.wikipedia.org/wiki/Root_of_unity_mod_n)*
8. *Root of unity.(http://en.wikipedia.org/wiki/Root_of_unity)*
9. *Convolutie. (http://nl.wikipedia.org/wiki/Convolutie)*
10. *Karatsuba-algorithm. (http://en.wikipedia.org/wiki/Karatsuba_algorithm)*
11. *Toom-Cook multiplication. (http://en.wikipedia.org/wiki/Toom-Cook_multiplication)*
12. *Divide&Conquer. (http//:en.wikipedia.org/wiki/Divide_and_conquer_algorithm)*
13. Richard Brent and Paul Zimmermann. *Modern Computer Arithmetic.* University Press, Cambridge. Section 1.3.6 Squaring, page 11 and Section 2.3.3 The Schönhage-Strassen algorithm, page 56.

Author: T.P.J. Kortekaas
Email: t.kortekaas@xs4all.nl
Web-pages: www.tonjanee.home.xs4all.nl