DIPLOMARBEIT

Fast Multiplication of Large Integers: Implementation and Analysis of the DKSS Algorithm

von

Christoph Lüders

chris@cfos.de

Institut für Informatik III

Prof. Dr. Michael Clausen

Rheinische Friedrich-Wilhelms-Universität Bonn

Bonn, 17. März 2015

rev. 3009

Ford: *"What do you get if you multiply six ... by nine* — by nine? *Is that it?"*
Arthur: *"That's it. Six by nine: forty-two! I always said that there is something fundamentally wrong about the universe."*

The Hitchhiker's Guide to the Galaxy radio series, episode 6

"Warum hat er es denn so eilig?"

N. N. about Arnold Schönhage and his fast multiplication

Acknowledgments

First and foremost, I wish to express my sincerest gratitude to my advisor Prof. Dr. Michael Clausen for his help, understanding, advice and encouragement. It was a pleasure to work under his guidance.

Furthermore, I wish to thank Karsten Köpnick and Nicolai Dubben for their proofreading and fruitful discussions. Their comments and questions were greatly appreciated.

I thank Prof. Dr. Arnold Schönhage for inspiring conversations many years ago when I first started to get interested in long numbers and also for his friendly comments on this thesis.

I am grateful to the authors of LaTeX for their excellent typesetting system and the authors of PGF/TikZ and PGFPlots for their packages to produce beautiful graphics. Furthermore, I thank the many contributors on tex.stackexchange.com for the fast and excellent help in times of need.

I thank Martin Winkler and our mutual business for understanding and support of my studies, especially when time got tight.

Many others have supported me, I just want to mention Dirk Eisenack, Heidi Förster, Goran Rasched and Susanne Röhrig. I am happy and grateful for their interest in, suggestions for and support of my work.

Also, I thank Anne-Sophie Matheron for asking me one too many times why I didn't finish my diploma.

Lastly, I thank my family and especially my mother for her boundless faith in me.

Contents

List of Figures

Symbols & Notation

$\mathbb{R}$	field of real numbers
$\mathbb{C}$	field of complex numbers
$\mathbb{Z}$	ring of all integers: $0, \pm 1, \pm 2, \ldots$
$\mathbb{N}$	$\{x \in \mathbb{Z} \mid x > 0\}$
$\mathbb{N}_0$	$\{x \in \mathbb{Z} \mid x \geq 0\}$
$[a : b]$	$\{x \in \mathbb{Z} \mid a \leq x \leq b\}$, for a, $b \in \mathbb{Z}$
$a/bc \cdot d$	$(a/(bc)) \cdot d$
$a \mid b$	$a \in \mathbb{Z}$ divides $b \in \mathbb{Z}$
$a \nmid b$	$a \in \mathbb{Z}$ does not divide $b \in \mathbb{Z}$
(a, b)	greatest common divisor of a, $b \in \mathbb{Z}$
$\lfloor a \rfloor$	$\max\{x \in \mathbb{Z} \mid x \leq a\}$, floor of $a \in \mathbb{R}$
$\lceil a \rceil$	$\min\{x \in \mathbb{Z} \mid x \geq a\}$, ceiling of $a \in \mathbb{R}$
$\log x$	logarithm of x to base 2
$\log^a x$	$(\log(x))^a$
$\exp_b^n(x)$	n-times iterated exponentiation of x to base b, cf. page 48
$\log^* x$	iterated logarithm of x to base 2, cf. page 48
a has degree-bound n	$\deg(a) < n$, a a polynomial, $n \in \mathbb{N}$
$g(n) \in O(f(n))$	$\exists c \geq 0, n_0 \in \mathbb{N}_0 : \forall n \geq n_0 : \lvert g(n) \rvert \leq c \cdot f(n)$

Chapter 1

Introduction

Multiplication of integers is one of the most basic arithmetic operations. Yet, if numbers get larger, the time needed to multiply two numbers increases as well. The naive method to multiply requires $c \cdot N^2$ bit-operations to multiply numbers with N digits, where c is some constant.[†] For large numbers this process soon becomes too slow and faster means are desirable.

Fortunately, in the 1960s methods were discovered that lowered the number of operations successively until in 1971 Schönhage and Strassen [SS71] found a technique that only requires $O(N \cdot \log N \cdot \log\log N)$ bit-operations.[‡] This algorithm was the asymptotically fastest known method to multiply until in 2007 Fürer [Fü07] found an even faster way. *Asymptotically* means that the algorithm was the fastest, provided numbers are long enough. Elaborate algorithms often involve some costs for set-up that only pay off if the inputs are long enough.[§]

Fürer's algorithm inspired De, Kurur, Saha and Saptharishi to their multiplication method [DKSS08], published in 2008, and a follow-up paper [DKSS13], the latter being discussed in this thesis and which I call *DKSS multiplication*. Both Fürer's and DKSS' new algorithms require $N \cdot \log N \cdot 2^{O(\log^* N)}$ bit-operations, where $\log^* N$ (pronounced "log star") is the number of times the logarithm function has to be applied to get a value ≤ 1.

However, Fürer conjectured that his new method only becomes faster than Schönhage and Strassen's algorithm for "astronomically large numbers" [Fü09, sec. 8]. Feeling unhappy about this vague assessment, I implemented the DKSS algorithm and compared it to Schönhage and Strassen's method to see if or when any improvement in speed could be achieved in practice. Both algorithms use only integer operations, in contrast to Fürer's algorithm that uses floating point operations.

The ability to multiply numbers with millions or billions of digits is not only academically interesting, but bears much practical relevance. For example, number theoretical tasks like primality tests require fast multiplication of potentially very large numbers. Such calculations can be performed nowadays with computer algebra systems like Magma, Maple, Mathematica, MATLAP, or Sage. Calculation of π or e to billions of digits or computing billions of roots of

[†] Usually, the constant is omitted and instead of $c \cdot N^2$ we write $O(N^2)$. The constant c is hidden in the $O(\ldots)$.

[‡] The logarithm function to base 10, $\log_{10} N$, is approximately the number of decimal digits of N. So if N is multiplied by 10, the logarithm just increases by 1. This is to show how slowly it is growing.

[§] Think of finding names in a stack of business cards: if you sort the cards first, you can find a name quickly, but it is only worth the effort if you search for a certain number of names.

Riemann's zeta function are other fields that requires fast large number multiplication [GG13, sec. 8.0].

Also, fast multiplication is an important building block of a general library for arithmetical operations on long numbers, like the GNU Multiple Precision Arithmetic Library [GMP14]. Addition and subtraction are not hard to implement and many of the more complex tasks — like inversion, division, square root, greatest common divisor — revert back to multiplication, cf. [GKZ07]. Once these operations are implemented for integers, they can be used to provide arbitrary-precision arithmetic for floating point numbers that attenuate rounding problems, cf. [GLTZ10].

Another big application for multiplication of long numbers is polynomial multiplication with integer coefficients, since it can be reduced to one huge integer multiplication through Kronecker-Schönhage substitution [Sch82, sec. 2]. If (multivariate) polynomials are of high degree, the resulting integers can become very long and fast means for multiplication are essential. Factoring of polynomials is also an important field of activity, see [GKZ07].

All elaborate multiplication methods use some sort of *fast Fourier transform* (FFT) at their core. The main idea behind all FFT multiplication methods is to break a long number into smaller pieces and interpret those pieces as coefficients of a polynomial. Since a polynomial of degree less than $2n$ is uniquely determined by its sample values for $2n$ pairwise different sample points, two polynomials of degree less than n can be multiplied like this:[†]

1. Evaluate both polynomials at the same $2n$ sample points,
2. multiply the sample values pairwise, and
3. interpolate the polynomial from the sample value products.

The FFT is "fast", since it computes n sample values with only $O(n \cdot \log n)$ operations, which is an enormous advance from the naive approach and its $O(n^2)$ operations. This method was already known by Gauss in 1805 [HJB85], but rediscovered by Cooley and Tukey in 1965 [CT65] and then revolutionized computation.

The method by Schönhage and Strassen breaks numbers of N bits into pieces of length $O(\sqrt{N})$ bits. Furthermore, it is cleverly designed to take advantage of the binary nature of today's computers: multiplications by 2 and its powers are particularly simple and fast to perform. This is why it has not only held the crown of the asymptotically fastest multiplication algorithm for over 35 years, but is also in widespread practical use today.

The new DKSS multiplication has a better asymptotic time bound, but its structure is more complicated. This elaborated structure allows input numbers to be broken into pieces only $O((\log N)^2)$ bits small. However, the arithmetic operations are more costly. The purpose of this thesis is to see if or when DKSS multiplication becomes faster than Schönhage-Strassen multiplication in practical applications.

Chapter 2 of this thesis presents an overview of multiplication algorithms from the naive method to techniques that provide a good trade-off if numbers are of medium length (like Karatsuba's

[†]Since the resulting polynomial is the product of its two factors, it has degree $2n - 2$. Therefore, at least $2n - 1$ sample points are needed to recover the result.

method in Section 2.4). The fast Fourier transform is introduced in Section 2.6 and is followed by a detailed description of Schönhage and Strassen's procedure in Section 2.9. All methods were implemented and their run-times are determined theoretically, measured in practice and illustrated graphically. Schönhage and Strassen's algorithm is more thoroughly analyzed in respect of its run-time, memory consumption and possible areas for improvement.

In Chapter 3 the DKSS algorithm is explained in detail and its run-time is analyzed theoretically. Section 3.4 describes the differences between my implementation and the paper [DKSS13].

Chapter 4 presents details of the implementation and illustrates its run-time (Section 4.4), memory requirements (Section 4.5) and source code complexity (Section 4.6) in comparison to Schönhage and Strassen's method both in numbers and graphics. Section 4.8 estimates the crossover point at which both algorithms become equally fast.

Lastly, Chapter 5 sums up the results and shows possible areas for improvement together with an assessment of their potential.

In this version of my thesis the typesetting has been modified to produce a more concise layout and some minor errors have been corrected.

Chapter 2

Overview of Established Algorithms

This chapter covers the well established algorithms to multiply large numbers, starting with the naive method. Methods for medium-sized numbers are discussed, the fast Fourier transform is introduced and Schönhage-Strassen multiplication is presented in detail. But first, some basic remarks about storage of large numbers and memory allocation for temporary storage are necessary.

2.1 Representation of Numbers

I assume my implementation is running on a binary computer and the machine has a native *word size* of w bits, so it can hold nonnegative integer values $0 \dots 2^w - 1$ in its general purpose registers. We call this unit a computer *word*. Today, the most common word sizes are 32 and 64 bits, therefore a machine register can hold integers between 0 and $2^{32} - 1 = 4\,294\,967\,295$ or $2^{64} - 1 = 18\,446\,744\,073\,709\,551\,615$, respectively.

If we want to do calculations with numbers that exceed the aforementioned range, we must use some multiple precision representation for them. If we call $W := 2^w$ the *wordbase*, we can write any nonnegative integer $a < W^n$ as $a = \sum_{i=0}^{n-1} a_i W^i$, with $a_i \in [0 : W - 1]$. We can view this as representing a with n words or "digits" in base W.

In my implementation a nonnegative number $a < W^n$ is represented by an array of n words as $a = (a_0, a_1, \dots, a_{n-2}, a_{n-1})$. The words are ordered with increasing indices in main memory. This ordering is called *little-endian*. It was a design choice to use this ordering: cache prefetching used to work better in forward loops, which are often used due to carry propagation. Modern CPUs seem to have remedied this problem.

The same ordering is used by Schönhage et al. [SGV94, p. 7] as well as in GMP, *The GNU Multiple Precision Arithmetic Library* [GMP14, sec. 16.1] and MPIR, a prominent GMP fork [MPIR12]. Interestingly, Zuras [Zur94] describes that storing numbers as big-endian worked better on his compiler and architecture.

The *i-code* in [SGV94, p. 6] stores the length n of the number after the most significant word in main memory. In contrast, my implementation keeps the length as a separate integer and provides both pointer to array and length as arguments on function calls.

Please note that we can usually pad any number with zero words on the upper end without influencing the result of operations (except for possible zero-padding of the result). It is a small waste of memory and processing time, but can simplify implementation of algorithms, for example, if an algorithm expects the length of a number to be even.

Negative numbers are represented as the two's complement of their absolute value. I followed the example of the i-code from [SGV94] in this design decision. It seems like a sensible choice, since execution time of simple operations like addition and subtraction benefit from this representation, whereas more elaborate operations like multiplication can afford the slight increase in execution time if negative numbers are being handled.

If negative numbers are handled and padding takes place, they have to be padded with all binary ones, that is, words with binary value -1. The most significant bit acts as sign bit if a number is interpreted as a signed value.

2.2 Memory Management

For all but the most basic functions we will need some temporary memory. To make routines fast, it is important that storage can be allocated and freed quickly. This forbids the use of the regular C-style `malloc()` or C++ `new` (which is just a wrapper for the former). C-style `malloc()` is designed to allow memory of different sizes to be allocated and freed at random and still maintain low fragmentation; many implementations are even thread-safe.

Since lifetime of temporary storage in our algorithms ends when a called function returns, we can use a stack-like model for temporary memory, which greatly simplifies the design of the allocator, makes it fast and doesn't need any locking. Plus, it has the added benefit of good cache locality. This is known as *region-based memory management*. In my code, this allocator is called `tape_alloc`.

To keep allocated memory continuous, every time memory is needed the allocator allocates more than is requested and records the total amount of memory allocated. When afterwards all memory is freed and later on a new allocation request is made, the allocator will right away allocate the total amount of memory used last time. The idea is that since algorithms often involve multiple calculations that handle long numbers in the same size-range, upcoming memory requirements will be as they were in the past.

Schönhage et al. implemented their algorithms on a hypothetical Turing machine called *TP* with six variable-length tapes and a special assembly language-like instruction set called *TPAL* (see [SGV94] and [Sch]). Of course, this machine has to be emulated on a real computer, so TPAL instructions are translated to C or assembly language for the target machine. Thus the tape-like structure of memory is retained.

The GMP library allocates temporary memory on the stack with `alloca()`. This should be fast and thread-safe, since no locking is required.

2.3 Ordinary Multiplication

All of us have learned to multiply with pencil and paper in school. This is often referred to as *ordinary multiplication* or *grade school multiplication*. The implementation of it is called `OMUL` (this name and others are inspired by [SGV94]).

Suppose we want to multiply two nonnegative integers a and b with lengths of n and m words, respectively, to compute the product $c := ab$ with length $n+m$. To do that we have to multiply each a_i, $i \in [0 : n-1]$ with each b_j, $j \in [0 : m-1]$ and add the product to c_{i+j}, which has to be set to zero before we start. Plus, there has to be some carry propagation.

In Python 3.x, our `OMUL` algorithm looks like this.[†] I have left out the carry propagation here, since this example only serves to show the principle. The C++ example will be more specific.

```
def omul(a, b):
    c = [0] * (len(a) + len(b))          # initialize result with zeros
    for i in range(0, len(a)):           # cycle over a
        for j in range(0, len(b)):       # cycle over b
            c[i+j] += a[i] * b[j]        # elementary mul and add
    return c
```

This Python implementation hides an important implementation detail: If a multiple precision number is made up of words and these are the same size as a processor register, then the product of two such words will be twice the size of a processor register! Our code must be able to handle this double-sized result. This is not a problem in the Python code above, since Python's `int` type is multiple precision by itself. A similar function in C++ shows more of that detail:

```
void omul(word* c, word* a, unsigned alen, word* b, unsigned blen) {
    memset(c, 0, (alen+blen) * sizeof(word));          // set result to zero
    for (unsigned i=0; i<alen; ++i) {                  // loop over a[i]'s
        word carry = 0;                                // for overflow
        unsigned j = 0;
        while (j < blen) {                             // loop over b[j]'s
            carry = muladdc(c[i+j], a[i], b[j], carry);
            ++j;
        }
        c[i+j] = carry;
    }
}
```

The type `word` is a placeholder for an unsigned integer type with the size of a processor word or smaller. The interesting part happens in the function `muladdc()`: `a[i]` and `b[j]` get multiplied, the input carry `carry` and the already computed result `c[i+j]` are added to the product, the lower part of the result is written back to memory (into `c[i+j]`) and the higher part of the result is saved in `carry` to be handled in the next iteration.

We have not yet addressed the problem of the double-sized multiplication result. We have two choices here: either use a `word` type that is only half the processor word size, so the product can be stored in a full processor word, or use some special function to get both the high and low part of a full sized multiplication in two separate variables. Luckily, modern compilers offer an intrinsic function for that and compile good code from it. The other option is still available, but takes about 60 % more time here for inputs of the same bit-size.[‡]

[†] The coding style is very un-pythonic and should only serve for explanation.

[‡] All timings are expressed in processor cycles and were done on an Intel Core i7-3770 CPU in 64-bit mode running Windows 7. Appendix A describes the test setup in detail.

For a 64-bit `word` type in Microsoft Visual C++, the `muladdc()` function looks like this:

```
typedef unsigned __int64 uint64;                    // keep it short
uint64 muladdc(uint64& mem, uint64 a, uint64 b, uint64 carry_in) {
   uint64 hiprod;
   uint64 lowprod = _umul128(a, b, &hiprod);        // intrinsic function
   hiprod += addc(mem, lowprod, carry_in);
   return hiprod;                                   // carry out
}

uint64 addc(uint64& mem, uint64 v, uint64 carry_in) {
   uint64 r1 = mem + v;
   uint64 carry_out = r1 < v;                       // overflow?
   uint64 r2 = r1 + carry_in;
   carry_out += r2 < carry_in;                      // overflow?
   mem = r2;
   return carry_out;
}
```

Again, we have to wrestle with word size limitation when handling overflow from addition in `addc()`. Unfortunately, Microsoft's C++ compiler doesn't offer a way to read the processor's carry flag. So, we have to do an additional comparison of the result with one of the inputs to determine overflow [War02, p. 29]. The resulting code is surprisingly fast, despite the superfluous comparison.

The total run-time of `OMUL` is easily determined: we have to do nm word-to-doubleword multiplications, since each a_i has to be multiplied by each b_j. The number of additions depends on the implementation: the Python version has nm additions, but they are at least triple-size, since the carries accumulate. The C++ version has four word-sized additions per multiplication.

In either case, the total run-time is $O(nm)$, and assuming $m = n$ it is $O(n^2)$.

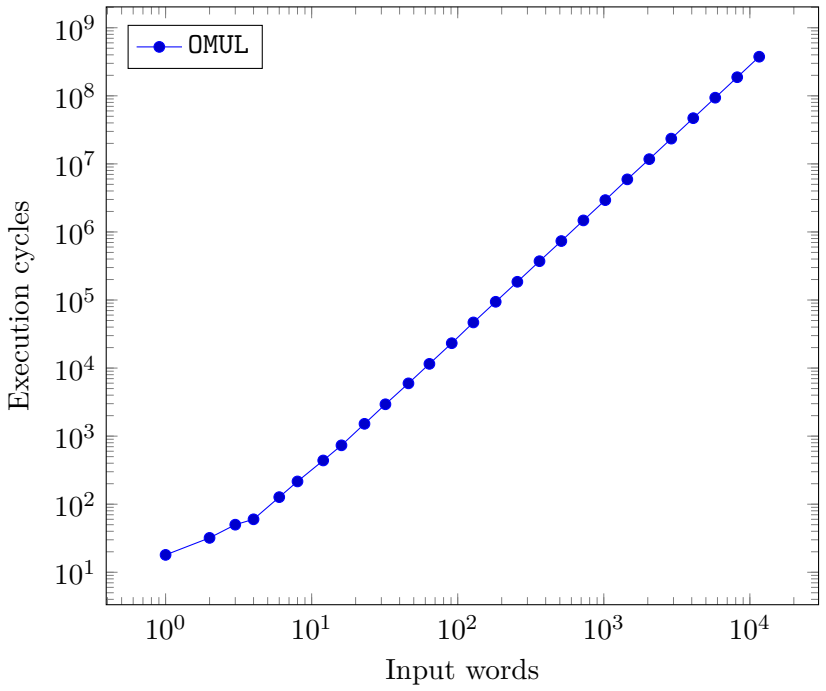

FIGURE 1: Execution time of `OMUL`

This is the "classical" time bound and even in 1956 it was still conjectured to be optimal, since no one had found a faster way to multiply for more than four millennia [Kar95].

Figure 1 shows a double-logarithmic graphical display of execution times in processor cycles for different input sizes. Observe the slight bend at the beginning, which shows the constant costs of calls and loop setup. Apart from that the graph is very straight, which shows that caching has no big influence, even though the highest input sizes well exceed the level 1 and level 2 cache sizes on the test machine.

2.4 Karatsuba Multiplication

Let $a, b < W^{2n}$ be two nonnegative integers, that is, both numbers consist of maximum $2n$ words. We are looking for a faster way to multiply both numbers to get their product $c = ab < W^{4n}$.

We can "cut" both numbers in half, that is, express them as

$$a = a_0 + a_1 W^n \quad \text{and} \quad b = b_0 + b_1 W^n, \tag{2.1}$$

with a_0, a_1, b_0, $b_1 < W^n$. The classical approach to calculate the full product from its four half-sized inputs is

$$\begin{aligned} ab &= (a_0 + a_1 W^n)(b_0 + b_1 W^n) \\ &= a_0 b_0 + (a_0 b_1 + a_1 b_0) W^n + a_1 b_1 W^{2n}. \end{aligned} \tag{2.2}$$

This way, we can break down a single $2n$-word multiplication into four n-word multiplications. Unfortunately, we don't gain any speed by this.

In 1960 Karatsuba found a faster way to multiply long numbers [KO63]. The following slightly improved version is due to Knuth [Knu97b, p. 295]. The implementation of it is called `KMUL`.

First, we compute the following three n-word multiplications

$$\begin{aligned} P_0 &= a_0 b_0 \\ P_1 &= (a_0 - a_1)(b_0 - b_1) \\ P_2 &= a_1 b_1 \end{aligned}$$

and use these three "small" products to recover the full product with only some extra additions and subtractions plus shifts (multiplications by powers of W):

$$\begin{aligned} ab &= P_0(1 + W^n) - P_1 W^n + P_2(W^n + W^{2n}) \\ &= a_0 b_0 (1 + W^n) - (a_0 - a_1)(b_0 - b_1) W^n + a_1 b_1 (W^n + W^{2n}) \\ &= a_0 b_0 + a_0 b_0 W^n - a_0 b_0 W^n + a_0 b_1 W^n + a_1 b_0 W^n - a_1 b_1 W^n + \\ &\quad a_1 b_1 W^n + a_1 b_1 W^{2n} \\ &= a_0 b_0 + (a_0 b_1 + a_1 b_0) W^n + a_1 b_1 W^{2n}. \end{aligned} \tag{2.3}$$

□

It looks like more work, but it is a real improvement. Since ordinary multiplication runs in $O(n^2)$, saving multiplications at the cost of additions, subtractions and shifts, which can be done in linear time, is a good deal in itself. But if we use Karatsuba's algorithm recursively, we can even achieve a time bound of $O(n^{\log 3}) \approx O(n^{1.585})$.

We are going to prove this bound by induction. Denote $T(n)$ the time it takes to multiply two n-word numbers. We know that we can reduce a $2n$-word multiplication to three n-word multiplications and some operations with linear run-time. Furthermore, we have to assign some cost to $T(1)$. So

$$\begin{aligned} T(1) &= c, \\ T(2n) &= 3T(n) + 2cn. \end{aligned}$$

We are going to show that

$$T(n) = 3cn^{\log 3} - 2cn.$$

This proof originates from [AHU74, p. 63]. It is easy to check the induction basis: $T(1) = 3c \cdot 1^{\log 3} - 2c \cdot 1 = c$. Next, we have to check the induction step:

$$\begin{aligned} T(2n) &= 3T(n) + 2cn \\ &= 3(3cn^{\log 3} - 2cn) + 2cn \\ &= 3c(3n^{\log 3}) - 6cn + 2cn \\ &= 3c(3n^{\log 3}) - 2c(2n) \\ &= 3c(2^{\log 3} n^{\log 3}) - 2c(2n) \\ &= 3c(2n)^{\log 3} - 2c(2n). \end{aligned}$$

□

If we implement this procedure, we first compute the three products P_0, P_1, P_2 and then use (2.3) to shift and add up the small products to get the full product. That means, we need some temporary storage for the small products and for the two factors that make up P_1.

To compute the two factors $(a_0 - a_1)$ and $(b_0 - b_1)$ we would like to avoid working with negative numbers, to keep things simple. To that end I use a knack (borrowed from GMP) and compare the minuend and subtrahend of the subtraction, always subtract the smaller from the larger and keep the sign bit in an extra variable. The implementation accommodates for the sign bit later when it re-assembles the three sub-products.

The mentioned ideas look like this when coded in C++:

```
void kmul(word* r, word* a, unsigned alen, word* b, unsigned blen) {
   if (alen < blen) {                              // b must not be longer than a
      swap(a, b),                                        // swap pointers
      swap(alen, blen);
   }
   if (blen < kmul_thresh) {                              // inputs too short?
      omul(r, a, alen, b, blen);                          // use omul
      return;
   }
   unsigned llen = blen / 2;                              // low part length
   unsigned ahlen = alen - llen;                          // a high part length
   unsigned bhlen = blen - llen;                          // b high part length

   // compute r0 = a0 * b0: this will lie in 'r' on index 0..llen-1
   kmul(r, a, llen, b, llen);
   // compute r2 = a1 * b1: this will lie in 'r' on index 2*llen..alen+blen-1
   kmul(r+2*llen, a+llen, ahlen, b+llen, bhlen);

   // allocate temporary space for differences and third mul
   tape_alloc tmp(4*ahlen + 1);
   word* sa = tmp.p;
   word* sb = tmp.p + ahlen;
   word* ps = tmp.p + 2*ahlen;

   // subtract values for later multiplication
```

```
    bool asign = compu_nn(a+llen, ahlen, a, llen) < 0;     // asign set if a1 < a0
    if (asign) subu(sa, ahlen, a, llen, a+llen, ahlen);    // a0 - a1 > 0
    else subu(sa, ahlen, a+llen, ahlen, a, llen);          // a1 - a0 >= 0

    bool bsign = compu_nn(b+llen, bhlen, b, llen) < 0;     // bsign set if b1 < b0
    if (bsign) subu(sb, ahlen, b, llen, b+llen, bhlen);    // b0 - b1 > 0
    else subu(sb, ahlen, b+llen, bhlen, b, llen);          // b1 - b0 >= 0

    // multiply both absolute differences
    unsigned plen = 2*ahlen + 1;                           // there can be a carry
    kmul(ps, sa, ahlen, sb, ahlen);
    ps[plen-1] = 0;

    // compute middle result
    if (asign == bsign) subu_on_neg(ps, plen, r, 2*llen); // ps = r0 - ps
    else addu_on(ps, plen, r, 2*llen);                     // ps += r0
    addu_on(ps, plen, r + 2*llen, ahlen + bhlen);          // ps += r2
    // add the final temp into the result
    addu_on(r+llen, ahlen + blen, ps, plen);
}
```

The code checks if input sizes suggest `OMUL` will be faster and if so, calls it instead. This is because `KMUL` is *asymptotically* faster than `OMUL`, but not so for small input lengths. Obviously, `KMUL` is more complicated than `OMUL`, as it uses several calls to add and subtract, conditional branches and temporary memory. All this takes its time compared to a very streamlined double-loop structure of `OMUL` that modern processors are really good at executing.

To achieve maximum performance we have to find the input length where `KMUL` starts to be faster than `OMUL`. This is called the *crossover point* or *threshold value*. The crossover point depends on the processor architecture, memory speed and efficiency of the implementation. To find the crossover point we have to benchmark both algorithms against one another at various input lengths.

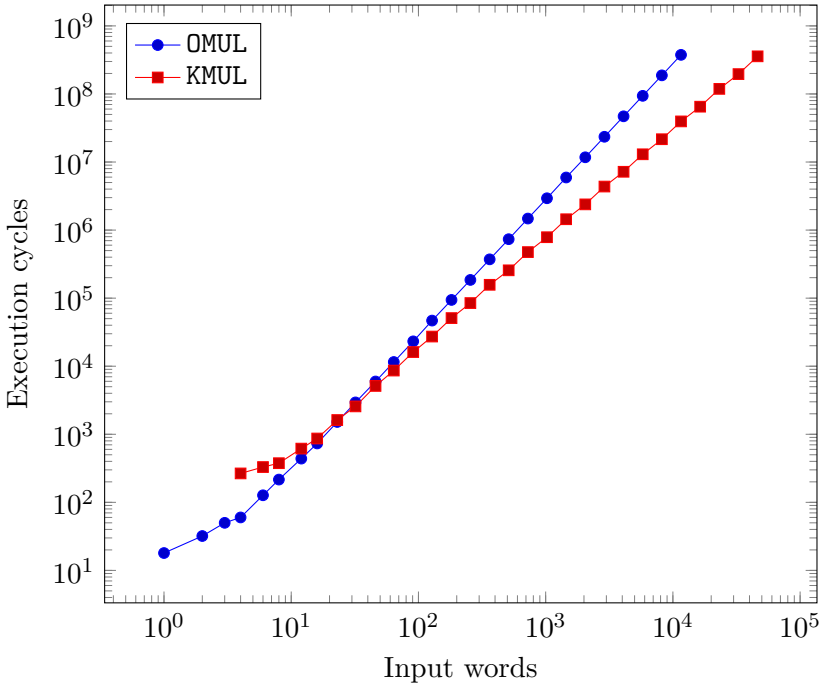

FIGURE 2: Execution time of `KMUL`

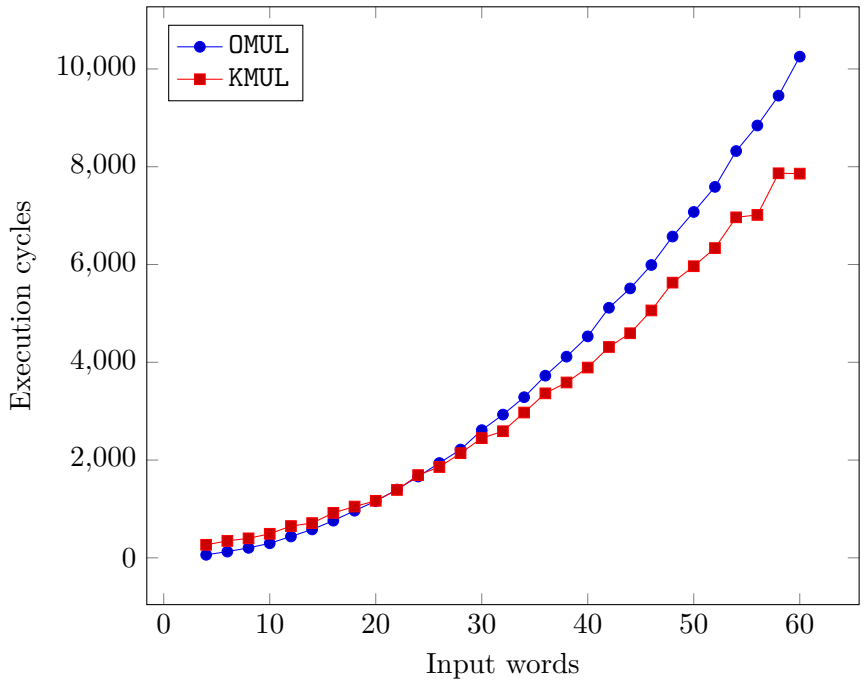

FIGURE 3: Execution time of `KMUL` (close-up)

Figure 2 shows the timings of `OMUL` and `KMUL`. We can see that KMUL is faster the longer the inputs are (with an input length of 10 000 words `KMUL` is about nine times faster than `OMUL`), but in the low ranges it is slower than `OMUL`.

To have a better look at the crossover point, Figure 3 has linear scaling and shows only input sizes up to 60 words. From the graph we can see the crossover point is at about 24 words input length, that is, about 460 decimal digits.

2.5 Toom-Cook Multiplication

Let us take a broader look at Karatsuba's algorithm: it follows from the fundamental theorem of algebra that any polynomial $a(x)$ of degree $< k$ is determined by its values at k distinct points. In the case of Karatsuba's algorithm, if we substitute W^n in (2.1) with the indeterminate x we get input polynomials of degree one: $a(x) = a_0 + a_1 x$ and $b(x) = b_0 + b_1 x$. If we multiply both, the result $c(x) \coloneqq a(x)b(x)$ is a polynomial of degree two.

What we did in Karatsuba multiplication can be understood as evaluating both polynomials $a(x)$ and $b(x)$ at points $\{0, -1, \infty\}$.[†,‡] Then we multiplied the results pointwise and interpolated to regain the polynomial $c(x)$. To regain the integer result we evaluated $c(x)$ at $x = W^n$.

We can generalize this technique: evaluate polynomials of degree $< k$ at $2k - 1$ distinct points, multiply pointwise and interpolate. The time bound of this scheme is $O(n^{\log_k(2k-1)})$, so for $k = 3$, 4, 5 it is approximately $O(n^{1.465})$, $O(n^{1.404})$ and $O(n^{1.365})$, respectively. This method is due to Toom [Too63] and Cook [Coo66].

[†]By abuse of notation $a(\infty)$ means $\lim_{x\to\infty} a(x)/x$ and gives the highest coefficient.

[‡]Other distinct points of evaluation would have done as well. For example, Karatsuba's original paper used $\{0, 1, \infty\}$.

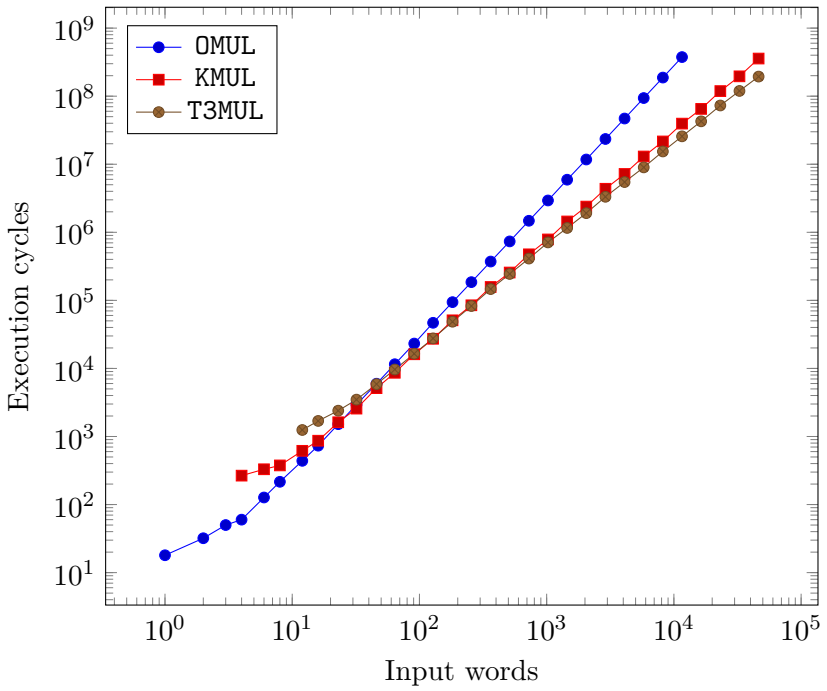

FIGURE 4: Execution time of `T3MUL`

The points for evaluation can be freely chosen (as long as they are distinct), but it is not obvious which choice leads to the simplest formulas for evaluation and interpolation. Zuras [Zur94] and Bodrato [BZ06] offer good solutions.

I implemented the Toom-Cook 3-way method from [BZ06] and called it `T3MUL`. Figure 4 shows a graph of execution time vs. input length. The crossover point of `T3MUL` and `KMUL` is at about 100 words or 2000 decimal digits.

Unfortunately, the exponent in the time bound drops slowly as k increases and the number of linear operations (additions, subtractions and divisions by constants) rises quickly with k. This leads to ever higher crossover points. Furthermore, each new k-way method has to be set in code individually. This calls for a more general solution.

2.6 The Fast Fourier Transform

We are going to have a look at the fast Fourier transform (or *FFT*) which was (re-)discovered in 1965 by Cooley and Tukey [CT65]. By choosing to evaluate the polynomial at certain special points it allows us to do the evaluation very quickly.

This is just a short description of the fast Fourier transform as far as it concerns us now. A good introduction can be found in Cormen et al. [CLRS09, ch. 30], Clausen and Baum [CB93] cover the topic from a group-theoretic standpoint and Duhamel and Vetterli [DV90] present a good overview of the plethora of different FFT algorithms.

Let R be a commutative ring with unity and let n be a power of 2.[†] The number n is called the *FFT length*. Let ω_n be a primitive n-th root of unity in R, that is, $\omega_n^n = 1$ and $\omega_n^k \neq 1$ for $k \in [1 : n-1]$. We simply write ω instead of ω_n, if the value of n is clear from the context. Furthermore, let $a(x) = \sum_{j=0}^{n-1} a_j x^j$ be a polynomial over R with degree-bound n.[‡]

For example, $\mathbb{R}$ contains only a primitive 2nd root of unity, namely -1, but no higher orders. But $\mathbb{C}$ does: $\omega_n = e^{2\pi i/n}$ is a primitive n-th root of unity in $\mathbb{C}$.

Another example is the quotient ring $\mathbb{Z}/n\mathbb{Z}$: it can be identified with the integers from 0 to $n-1$, where all operations are executed modulo n. $\mathbb{Z}/n\mathbb{Z}$ can contain up to $n-1$ roots of unity. In the case of $n = 17$, $\omega = 3$ is a primitive 16^{th} root of unity.

We want to evaluate $a(x)$ at n distinct, but otherwise arbitrary points. If we choose to evaluate $a(x)$ at ω^k, $k \in [0 : n-1]$, we can design the evaluation particularly efficient. Because ω is a *primitive* n-th root of unity, we know ω^0, ω^1, …, ω^{n-1} to be pairwise different.

We can re-sort $a(x)$ in even and odd powers and rewrite it as

$$\begin{aligned} a(x) &= a_0 + a_1 x + a_2 x^2 + a_3 x^3 + \dots \\ &= a_0 + a_2 x^2 + \dots + a_1 x + a_3 x^3 + \dots \\ &= \underbrace{a_0 + a_2 x^2 + \dots}_{=:e(x^2)} + x\underbrace{(a_1 + a_3 x^2 + \dots)}_{=:o(x^2)} \\ &= e(x^2) + x o(x^2), \end{aligned}$$

where $e(x)$ and $o(x)$ are polynomials with half the degree-bound as $a(x)$. Since n is a power of 2, we can proceed recursively with this divide-and-conquer approach until the degree-bound of both polynomials $e(x)$ and $o(x)$ is one, that is, both consist only of a constant.

We can evaluate $a(x)$ at ω^k, $k \in [0 : n/2 - 1]$ and get

$$a(\omega^k) = e(\omega^{2k}) + \omega^k o(\omega^{2k}).$$

But note what we get if we evaluate $a(x)$ at $\omega^{k+n/2}$, $k \in [0 : n/2 - 1]$:

$$\begin{aligned} a(\omega^{k+n/2}) &= e((\omega^{k+n/2})^2) + \omega^{k+n/2} o((\omega^{k+n/2})^2) \\ &= e(\omega^{2k+n}) + \omega^{k+n/2} o(\omega^{2k+n}) \\ &= e(\omega^{2k}) - \omega^k o(\omega^{2k}), \end{aligned}$$

since $\omega^{n/2} = -1$ and $\omega^n = 1$.

If we have already computed $e(\omega^{2k})$ and $o(\omega^{2k})$ we save time by calculating both $a(\omega^k)$ and $a(\omega^{k+n/2})$ side by side:

$$\begin{aligned} a(\omega^k) &= e(\omega^{2k}) + \omega^k o(\omega^{2k}) \\ a(\omega^{k+n/2}) &= e(\omega^{2k}) - \omega^k o(\omega^{2k}). \end{aligned}$$

This is the concept that makes the fast Fourier transform efficient. After solving both halves of the problem we can calculate two results in $O(1)$ additional time.[§]

[†] Please note that n no longer designates an input length in words.

[‡] Any integer $n > \deg(a)$ is called a *degree-bound* of a.

[§] The simultaneous calculation of sum and difference is called a *butterfly operation* and the factors ω^k in front of $o(\omega^{2k})$ are often called *twiddle factors*.

There are different types of FFTs and the one just described is called a *Cooley-Tukey FFT* of length n. More precisely, it is a radix-2 decimation in time FFT. See [DV90] for other types of FFTs.

We can write this algorithm as a recursive function in Python. The computation of the actual root of unity has been left out of this example to keep it independent of the ring R.

```
def fft(a):                                  # a is a list
    n = len(a)                               # degree-bound of a
    if n <= 1: return a
    even = fft(a[0::2])                      # slice up even numbered values
    odd = fft(a[1::2])                       # ... and odd
    r = [0] * n                              # fill list with n dummies
    for k in range(0, n//2):                 # n//2 means integer divide
        w = root_of_unity(n, k)              # n-th root to k-th power
        r[k] = even[k] + w * odd[k]
        r[k+n//2] = even[k] - w * odd[k]
    return r
```

Since at each recursion level the input list is split into values with even and odd indices, we get the structure shown in Figure 5, if we assume a start with eight input values.

Notice the ordering of the indices at the lowest level: the values are at an index which is the *bit-reversed* input index. "Bit-reversed" here means only reversing the bits that are actually used in indexing: in the last example we had eight values, hence we needed 3 bits for indexing. Accordingly, the bit-reversed index of, for example, $a_3 = a_{011_b}$ is $110_b = 6$.

The bit-reversal is a consequence of the splitting of the array into even and odd indices. Since even indices have the lowest bit set to zero, all "left" members of the output array have the highest bit of the index set to zero, whereas all "right" members have odd indices and have the highest bit set to one. This repeats itself through all levels.

We use this observation to decrease the memory footprint: the `fft()` function listed above uses temporary memory at each level, first to split up the input in even and odd indexed values and then to create the list of return values. We would like to save those allocations. Luckily, that is possible. The following design and the term "shuffle" is taken from Sedgewick [Sed92, ch. 41].

If we reorder the input list according to its bit-reversed indices, all even indexed values are in the first half and all odd indexed values in the second half. This saves us the creation of function arguments for the recursive calls. All we need to hand over to the lower levels is the position

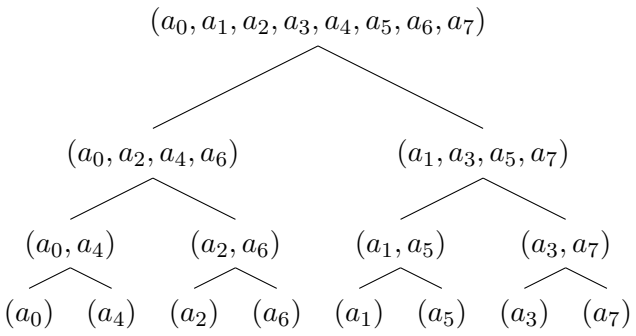

FIGURE 5: Splitting an array into even and odd positions

and length in the array they should work on, since the values are already in the right order. Then the tree of function arguments looks like Figure 6.

We don't even need extra storage for the return values! We can use the memory of the input parameters and overwrite it with the return values; the input parameters are no longer needed after the function has calculated the return values from them.

If we put all this into code, our Python function looks like this:

```
def bit_rev(x, b):                          # reverse b lower bits of x
   return sum(1<<(b-1-i) for i in range(0, b) if (x>>i) & 1)

def shuffle(a):                             # shuffle input list a
   r = []                                   # empty list
   b = (len(a)-1).bit_length()              # bits used for indexing
   pos = [bit_rev(n, b)
          for n in range(0, len(a))]        # list of new positions
   for i in pos:                            # cycle through list of positions
      r.append(a[i])                        # ... and build return list
   return r

def fft_eval(a, pos, n):                    # work on a[pos..pos+n-1]
   half = n//2                              # integer divide
   if half > 1:
      fft_eval(a, pos, half)                # even part
      fft_eval(a, pos+half, half)           # odd part
   for k in range(0, half):
      w = root_of_unity(n, k)               # n-th root to k-th power
      t = w * a[pos+half+k]                 # multiply only once
      a[pos+half+k] = a[pos+k] - t          # must use this order
      a[pos+k] = a[pos+k] + t               # ... to avoid overwriting
   return

def fft_inplace(a):
   aa = shuffle(a)                          # create reordered a
   fft_eval(aa, 0, len(aa))                 # fft works in-place
   return aa
```

Let us analyze the number of arithmetic operations of the algorithm above. We have assumed that n is a power of 2. With each level the length of the input is halved until $n = 1$; this leads to $\log n$ levels of recursion. Furthermore, while the number n gets halved with each level, both halves are worked on, so all values are cycled over (see Figure 6). Since two return values are calculated with three arithmetic operations (two additions and one multiplication), the arithmetic cost per level is $3n/2$, which leads to a total cost for the whole operation of $T(n) = 3n/2 \cdot \log n$.

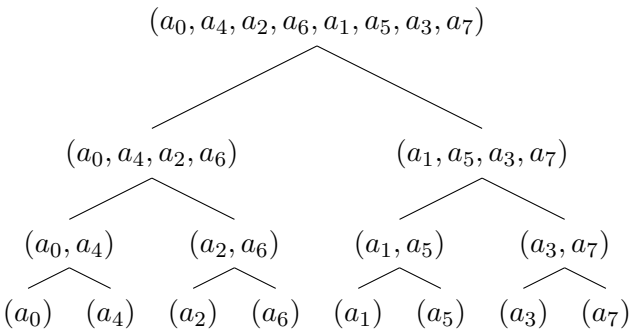

FIGURE 6: Halving the already shuffled array

We can prove the run-time more formally (inspired by [Sed92, pp. 77–78]). Obviously, $T(1) = 0$. Then the total arithmetic cost is

$$\begin{aligned} T(n) &= 2T(n/2) + 3n/2 \\ &= 2(2T(n/4) + 3n/2) + 3n/2 \\ &= 4T(n/4) + 3n/2 + 3n/2 \\ &\vdots \\ &= 2^{\log n} T(n/2^{\log n}) + 3n/2 \cdot \log n \\ &= nT(1) + 3n/2 \cdot \log n \\ &= 3n/2 \cdot \log n. \end{aligned} \tag{2.4}$$

2.7 FFT-based Polynomial Multiplication

Now that we have introduced the fast Fourier transform and proved its run-time, let us see how we can use it to multiply two polynomials rapidly.

Let R be a commutative ring with unity and let n be a power of 2. Let ω be a primitive n-th root of unity in R. Furthermore, let $a(x) = \sum_{j=0}^{n/2-1} a_j x^j$ and $b(x) = \sum_{j=0}^{n/2-1} b_j x^j$ be polynomials over R.

Please note that the polynomials $a(x)$ and $b(x)$ have a degree-bound of $n/2$. Since we are about to compute $c(x) := a(x)b(x)$ we need to choose the number of sample points n as $n > \deg(c) = \deg(a) + \deg(b)$. To keep notation simple, we let $a_j = b_j = 0$ for $j \in [n/2 : n-1]$.

We evaluate both input polynomials at sample points ω^k, $k \in [0 : n-1]$ to get sample values $\widehat{a}_k := a(\omega^k)$ and $\widehat{b}_k := b(\omega^k)$. Then, we multiply the sample values pairwise to get the $\widehat{c}_k := \widehat{a}_k \widehat{b}_k$. But how to retrieve the result polynomial $c(x)$ from the $\widehat{c}_k$? We will see how to accomplish that with ease if R, n and ω meet two additional requirements:

- $\omega^k - 1$, $k \in [1 : n-1]$, must not be a zero divisor in R, and (2.5)
- n must be a unit in R, meaning n is invertible. (2.6)

We return to these requirements later. Assuming that they hold, we can prove that the same algorithm can be used on the $\widehat{c}_k$ to regain the c_j that was used to compute the $\widehat{a}_k$ and $\widehat{b}_k$ in the first place! That is to say: the Fourier transform is almost self-inverse, except for ordering of the coefficients and scaling.

Let us see what happens if we use the $\widehat{a}_k = a(\omega^k) = \sum_{j=0}^{n-1} a_j \omega^{kj}$ as coefficients of the polynomial $\widehat{a}(x) := \sum_{k=0}^{n-1} \widehat{a}_k x^k$ and evaluate $\widehat{a}(x)$ at ω^ℓ, $\ell \in [0 : n-1]$, to compute $\widehat{\widehat{a}}_\ell := \widehat{a}(\omega^\ell)$. We get

what is called an *inverse transform*:

$$\begin{aligned}\widehat{\widehat{a}}_\ell &= \widehat{a}(\omega^\ell) \\ &= \sum_{k=0}^{n-1} \widehat{a}_k \omega^{\ell k} \\ &= \sum_{k=0}^{n-1} \Big(\sum_{j=0}^{n-1} a_j \omega^{kj} \Big) \omega^{\ell k} \\ &= \sum_{j=0}^{n-1} \sum_{k=0}^{n-1} a_j \omega^{(j+\ell)k} \\ &= \sum_{j=0}^{n-1} a_j \sum_{k=0}^{n-1} (\omega^{j+\ell})^k \\ &= n \cdot a_{(-\ell) \bmod n}. \end{aligned}$$

□

The last line holds due to the sum of the geometric series:

$$\sum_{k=0}^{n-1} (\omega^{j+\ell})^k = \begin{cases} \dfrac{\omega^{(j+\ell)n} - 1}{\omega^{j+\ell} - 1} = 0 & \text{if } j + \ell \not\equiv 0 \pmod{n}, \quad (2.7) \\ \displaystyle\sum_{k=0}^{n-1} 1 = n & \text{if } j + \ell \equiv 0 \pmod{n}. \quad (2.8) \end{cases}$$

Now we see why (2.5) is required: if $\omega^k - 1$, for $k \in [1 : n-1]$, is a zero divisor we are not allowed to do the division in (2.7). Furthermore, to remove the factor n in front of every $a_{(-\ell) \bmod n}$ we need (2.6).

If we want to get the $\widehat{\widehat{a}}_\ell$ in the same order as the original a_j, we can simply evaluate $\widehat{a}(x)$ at $\omega^{-\ell}$ instead of ω^ℓ. This is called a *backwards transform*.

To summarize: we can evaluate the $\widehat{c}_k$ at points $\omega^{-\ell}$ to retrieve $n \cdot c_\ell$, divide by n and have thus recovered the coefficients of our desired product polynomial.

The overall arithmetic cost of this polynomial multiplication method is three FFTs in $O(n \log n)$ plus n multiplications of pairs of sample values in $O(n)$ plus n normalizations in $O(n)$. The FFTs dominate the total cost, so it is $O(n \log n)$.

2.8 Modular FFT-based Multiplication

We can put last section's method into action and design a fast multiplication algorithm for long numbers using the quotient ring $R = \mathbb{Z}/p\mathbb{Z}$, with prime p. This is sometimes called a *number theoretic transform* or *NTT*. According to [Knu97b, p. 306] this method goes back to Strassen in 1968.

We want to multiply nonnegative integers a and b to get the product $c \coloneqq ab$. We are free to choose an arbitrary p for our calculations, as long as last section's requirements are met. Furthermore, our choice of p should be well suited for implementation. If we choose p to be

prime it means that the ring $\mathbb{Z}/p\mathbb{Z}$ is even a field, so we are sure to meet requirements (2.5) and (2.6). This field is denoted $\mathbb{F}_p$.

For the FFTs we need roots of unity of sufficiently high degree. Let $p > 3$. Since p is prime, we know that $\mathbb{F}_p^* = \{1, 2, 3, \ldots, p-1\}$ is a cyclic multiplicative group of order $p-1$. We call $g \in \mathbb{F}_p^*$ a *generator* of $\mathbb{F}_p^*$ if its powers g^j, $j \in [0 : p-2]$, create the whole group. Also, g is a primitive $(p-1)$-th root of unity. Note that $g^{p-1} = g^0 = 1$.

$\mathbb{F}_p^*$ contains $p-1$ elements. Since $p-1$ is even, we can find integers $u, v > 1$, such that $p-1 = uv$. Without loss of generality, let v be a power of 2. We know that $g^{p-1} = 1$, hence $g^{uv} = (g^u)^v = 1$. Since v divides $p-1$, we know g^u is a v-th primitive root of unity in $\mathbb{F}_p^*$.

Let us see how to reduce a long multiplication to polynomial multiplication: we have to distribute the bits of input numbers a and b to coefficients of polynomials $a(x)$ and $b(x)$. In Karatsuba's algorithm we did cut the input numbers in two pieces of n words each, or wn bits, where w is the word size and $W = 2^w$ is the wordbase. Accordingly, evaluating polynomial $a(x)$ at W^n yielded number a. Now we are going to cut the input numbers into pieces of r bits. But how to choose r?

The larger r is, the less coefficients we get, that is, the lower the degree of the polynomial. In consequence, this can lead to smaller FFT lengths, which are faster to compute. This is why we want to choose r as large as possible.

If we multiply polynomials $a(x) = \sum_{j=0}^{n/2-1} a_j x^j$ and $b(x) = \sum_{k=0}^{n/2-1} b_k x^k$ to get product $c(x) \coloneqq a(x)b(x) = \sum_{\ell=0}^{n-2} c_\ell x^\ell$ with $c_\ell = \sum_{j+k=\ell} a_j b_k$, observe that c_ℓ can contain up to $n/2$ summands. By construction a_j, $b_k < 2^r$, hence $c_\ell < \frac{n}{2}(2^r)^2 = n2^{2r-1}$. But c_ℓ must also be less than p. Hence, our choice of p must make sure that

$$p \geq n2^{2r-1}. \tag{2.9}$$

For practical reasons, we want to choose a prime p that can be handled easily by the target machine's processor, hence I chose p to be almost as big as the wordbase, so it can still be stored in one machine word. "Almost" means $\lfloor \log p \rfloor = w - 1$ to maximize the use of available bits per word.

The above mentioned constrains led me to choose the following parameters:[†]

Word size (bits)	Modulus p	Composition of p	Generator g
8	193	$3 \cdot 2^6 + 1$	5
16	40 961	$5 \cdot 2^{13} + 1$	3
32	3 489 660 929	$13 \cdot 2^{28} + 1$	3
64	10 232 178 353 385 766 913	$71 \cdot 2^{57} + 1$	3

[†]I reproduce the numbers here, since it required some effort to calculate them. If one wants to do FFT with modular arithmetic, one must first find a suitable prime modulus p and a matching primitive n-th root of unity. So here they are.

From these numbers we can calculate the respective primitive n-th root of unity ω:

Word size (bits)	Order n of primitive root ω	Primitive n-th root ω
8	2^6	$5^3 = 125$
16	2^{13}	$3^5 = 243$
32	2^{28}	$3^{13} = 1\,594\,323$
64	2^{57}	$3^{71} = 3\,419\,711\,604\,162\,223\,203$

Now that we have chosen p, we can use (2.9) to calculate the maximum r for a given FFT length n:

$$\begin{aligned} n2^{2r-1} &\leq p \\ \log(n2^{2r-1}) &\leq \log p \\ \log n + 2r - 1 &\leq \log p \\ 2r &\leq \log p - \log n + 1 \\ r &\leq \frac{1}{2}(\log p - \log n + 1) \end{aligned}$$

Choosing r determines the degree of the polynomials and hence n, which in turn can have an influence on r. So, we might have to cycle several times over this formula to find the largest r and smallest n.

Please note that this also imposes an upper bound on the length of input numbers this algorithm can handle:

$$\begin{aligned} \log n + 2r - 1 &\leq \log p \\ \log n &\leq \log p - 2r + 1 \\ \log n &\leq w - 2r && \text{(since } \lfloor \log p \rfloor = w - 1\text{)} \\ \log n &\leq w - 2 && \text{(r has to be at least 1)} \\ n &\leq 2^{w-2} \\ n &\leq W/4. \end{aligned}$$

This determines the maximum FFT length. In this case, r was 1 bit and hence the maximum output length is $W/4$ bits or $W/32$ bytes. Choosing a larger r only makes matters worse. The maximum FFT length might be even less than that, since the order of ω limits the FFT length as well.

Now that we have chosen the necessary parameters, we can attend to the implementation. A Python version of the main routine looks pretty straightforward. I termed this function `QMUL`, alluding to QuickMul by Yap and Li [YL00].

```
def qmul(a, b):
    p, n, w, r = select_param(a, b)      # p: prime modulus, n: FFT length
                                         # w: n-th root, r: bits per coefficient
    al = split_input(a, p, r, n)         # split inputs into n parts, ...
    bl = split_input(b, p, r, n)         # ... each maximum r bits long

    al = shuffle(al)                     # shuffle inputs
    bl = shuffle(bl)

    fft_eval(al, 0, n, w)                # evaluate inputs at roots of unity
    fft_eval(bl, 0, n, w)
```

```
    cl = []                             # empty list
    for i in range(0, n):               # multiply pointwise
        cl.append(al[i] * bl[i])        # append to list

    cl = shuffle(cl)                    # shuffle result
    fft_eval(cl, 0, n, w)               # evaluate result

    inv = modinv(n, p)                  # normalize result
    for i in range(0, n):
        cl[i] *= inv

    c = reasm(cl, r)                    # reassemble result
    return c
```

The functions `fft_eval()` and `shuffle()` have already been shown. Functions `reasm()` and `split_input()` are new: they cut up the input numbers and add up the coefficients of the resulting polynomial to a number, respectively. To find the proper number of bits per coefficient `r` and compute the FFT length `n` and matching root `w` function `select_param()` is used.

The actual implementation I used for benchmarking was done in C++. To give an impression of the code, the following function is a simplified version of the evaluation. The actual code is more complicated, since I use C++ templates to unroll the last five levels of the FFT. This saves some call and loop overhead at the cost of code size.

```
void qmul_evaluate(word* p, unsigned i, unsigned lg) {
    unsigned n = 1 << lg;                   // number of values
    unsigned half = n/2;                    // half of them
    if (half > 1) {
        qmul_evaluate(p, i, lg-1);          // even part
        qmul_evaluate(p, i+half, lg-1);     // odd part
    }

    // w^0=1: no multiplication needed
    word t = p[i+half];
    p[i+half] = modsub(p[i], t);
    p[i] = modadd(p[i], t);

    // handle w^k, k>0
    word* pw = pre_w[lg];                   // use precomputed roots
    for (unsigned k=1; k<half; ++k) {
        word t = modmul(pw[k], p[i+half+k]);
        p[i+half+k] = modsub(p[i+k], t);
        p[i+k] = modadd(p[i+k], t);
    }
}
```

Functions `modadd()` and `modsub()` are pretty straightforward and I don't include them here, but `modmul()` is more complicated. It takes two 64-bit numbers as inputs and multiplies them modulo p. We recall that there is an intrinsic compiler function to do the multiplication, but the result has 128 bits and has to be reduced modulo p. To accomplish that, we could use a 128-by-64-bit division, but it is quite slow and takes up to 75 cycles.

Warren [War02, pp. 178–188] shows a solution how to replace a division by a constant with a multiplication and a shift. Since the input has 128 bits, the multiplication has to be 128-bit wide as well. But this only results in floor division. To get the rest from division we must do one more multiplication and one subtraction. In total, we have to do six 64-bit multiplications, plus some additions and shifts to do the one modular multiplication. Benchmarking shows that the whole modular multiplication can be done like that in about 10 cycles.

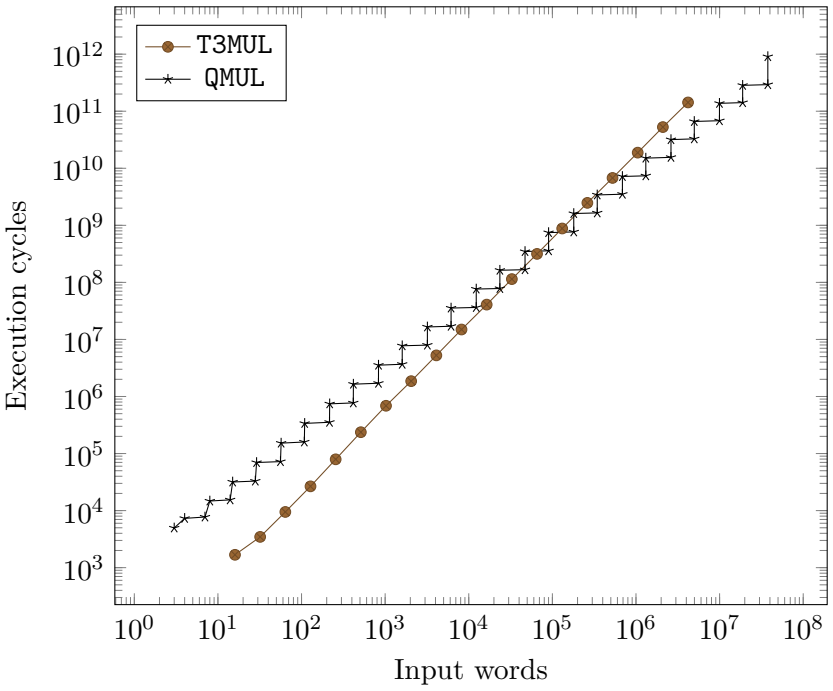

FIGURE 7: Execution time of `QMUL`

To save the time needed to calculate the roots of unity, I use arrays of precomputed roots `pre_w[lg]`. These powers are independent of the inputs and are reused every time `QMUL` is used.

A graph of execution cycles of `QMUL` in comparison to `T3MUL` is presented in Figure 7. Please note that this graph covers a wider range of sizes than the previous graphs. We see that our new algorithm is asymptotically faster than the hitherto used `T3MUL` implementation.

The stair-like shape of the graph is a result of the FFT: if input numbers get too large, the FFT depth must be increased by one level, thereby doubling the number of evaluation points. From these graphs we can say that `QMUL` starts to outperform `T3MUL` for inputs with a length of about 110 000 words or more, that is, about 2 100 000 decimal digits.

So we found an algorithm with a good asymptotic cost, but it only starts to pay off if inputs are quite long. Why is that so? What are the weak points of `QMUL`?

- The modular multiplication and reductions are expensive. Six word-sized multiplications are not cheap.
- The FFT length is large and hence many extra bits room for the sum of the coefficient products must be left free. Since the unit of operation is only a processor word, this "eats up" quite some percentage of its size. Plus, it implies a large FFT depth as well.
- The maximum length for long numbers is limited to $W/32$ bytes, even if larger numbers could be handled by the machine.

The following celebrated algorithm will address all of the weak points listed above.

2.9 Modular Schönhage-Strassen Multiplication

The idea of Schönhage-Strassen multiplication (of which my implementation is called `SMUL`) is to perform FFTs in rings of the form $\mathbb{Z}/(2^K+1)\mathbb{Z}$, which are sometimes called *Fermat rings*. In Fermat rings, 2 is a primitive $2K$-th root of unity. This fact can be exploited to speed up multiplications by roots of unity during the FFTs: multiplications by powers of 2 can be implemented as shifts followed by a modular reduction and thus take only $O(K)$ time. This is the cornerstone of the efficiency of Schönhage and Strassen's multiplication.

Since all roots of unity are powers of 2, we don't need to precompute them as in `QMUL`, but can just keep track of shift counts. Furthermore, modular reductions are simple and can be done with just another long number addition and/or subtraction. In this context, a shift followed by a modular reduction is called a *cyclic shift*.

`SMUL` always computes the product modulo 2^N+1, where N can be chosen. If we multiply input numbers a and b to get $c := ab$, we have to provide an N that is big enough to hold the full result. `SMUL` reduces a multiplication with N bits to smaller multiplications of $K \approx \sqrt{N}$ bits, in contrast to a reduction to word size as in `QMUL`.[†] If the size of pointwise multiplications exceeds a certain threshold, `SMUL` is used recursively, otherwise a simpler algorithm takes over.

This algorithm was first published by Schönhage and Strassen in 1971 [SS71] and provided results modulo 2^N+1, where N itself is a power of 2. A later version published by Schönhage [Sch82] relaxes the requirement to "suitable numbers" of the form $N = \nu 2^n$, $\nu \in [n-1 : 2n-1]$. For the implementation we can relax the requirement even more: Section 2.9.3 lists the details.

We introduce some notation: to compute the product c of nonnegative numbers a and b, we do FFTs in the ring $R := \mathbb{Z}/(2^K+1)\mathbb{Z}$. We use a Cooley-Tukey FFT and thus the FFT length n has to be a power of 2. Since we can choose R (and hence K) to suit our needs, we choose $K = r2^m$, with positive integers r and m. Our choice of K and m will in turn determine N, where $N = s2^m$, with positive integer s. This s is the number of input bits per coefficient.

It it easy to see that 2 is a primitive $2K$-th root of unity: since $2^K+1 \equiv 0$, we have $2^K \equiv -1$ and hence $2^{2K} \equiv 1$. Furthermore, it is obvious that for $u \in [1 : K-1]$ we get $2^u \not\equiv \pm 1$. For $K+v =: u \in [K+1 : 2K-1]$ we see that $2^u = 2^{K+v} = 2^K 2^v = -2^v \not\equiv 1$.

Because the FFT length is a power of 2, we need a primitive root of unity of the same order. Since 2 is a primitive root of unity of order $2K = 2r2^m$, it holds that $1 \equiv 2^{2K} = 2^{2r2^m} = (2^{2r})^{2^m}$. This makes $\omega := 2^{2r}$ a primitive 2^m-th root of unity and the FFT length $n := 2^m$. We deliberately chose an even exponent for ω, since we will be needing $\sqrt{\omega}$ later.

2.9.1 Invertibility of the Transform

For the existence of the inverse FFT requirements (2.5) and (2.6) have to be met. Since 2^K+1 may not be prime, we cannot rely on our argument from Section 2.8, so we must show that the requirements are met here, too:

- With $\omega = 2^{2r}$, $\omega^j - 1$, $j \in [1 : n-1]$, must not be a zero divisor in $\mathbb{Z}/(2^K+1)\mathbb{Z}$, and (2.10)

[†] A reduction to word size is usually not possible in `SMUL`, because the FFT length is not sufficiently large to cut input numbers in parts so small, since there are not enough roots of unity.

- $n = 2^m$ must be a unit in $\mathbb{Z}/(2^K + 1)\mathbb{Z}$. (2.11)

To prove (2.10) we need some identities about the *greatest common divisor* (gcd): Let a, b and u be positive integers and (a, b) signify the greatest common divisor of a and b. Then the following identities hold:

$$(a, b) = (b, a), \tag{2.12}$$
$$(a, b) = (a - b, b), \text{ if } a \geq b, \tag{2.13}$$
$$(ua, ub) = u(a, b), \tag{2.14}$$
$$(ua, b) = (u, b)(a, b), \text{ if } (u, a) = 1, \tag{2.15}$$
$$(ua, b) = (a, b), \text{ if } u \nmid b, \tag{2.16}$$
$$(2^a - 1, 2^b - 1) = 2^{(a,b)} - 1, \tag{2.17}$$
$$(2^a - 1, 2^a + 1) = 1, \tag{2.18}$$
$$(2^a - 1, 2^b + 1) = \frac{2^{(a,2b)} - 1}{2^{(a,b)} - 1}. \tag{2.19}$$

Identities (2.12) – (2.16) are well known, so we don't prove them here.

We prove (2.17) by induction on $a + b$. We assume without loss of generality that $a \geq b$. The induction basis is easily checked: $(2^1 - 1, 2^1 - 1) = 1 = 2^{(1,1)} - 1$.

Now we show the induction step: we assume $(2^\alpha - 1, 2^\beta - 1) = 2^{(\alpha,\beta)} - 1$, for $\alpha + \beta < a + b$. We use (2.13) and get

$$\begin{aligned}
(2^a - 1, 2^b - 1) &= (2^a - 1 - (2^b - 1), 2^b - 1) \\
&= (2^a - 2^b, 2^b - 1) \\
&= (2^b(2^{a-b} - 1), 2^b - 1) \\
&= (2^{a-b} - 1, 2^b - 1) && \text{(by (2.16))} \\
&= 2^{(a-b,b)} - 1 && \text{(by IH)} \\
&= 2^{(a,b)} - 1. && \text{(by (2.13))}
\end{aligned}$$

□

To prove (2.18) we use (2.13) and see that

$$\begin{aligned}
(2^b - 1, 2^b + 1) &= (2^b - 1, 2^b + 1 - (2^b - 1)) \\
&= (2^b - 1, 2) \\
&= 1.
\end{aligned}$$

□

To prove (2.19) we use the well known difference of squares $a^2 - b^2 = (a + b)(a - b)$ and apply it to our case, where it yields $2^{2b} - 1 = (2^b + 1)(2^b - 1)$. It holds that

$$\begin{aligned}
2^{(a,2b)} - 1 &= (2^a - 1, 2^{2b} - 1) && \text{(by (2.17))} \\
&= (2^a - 1, (2^b + 1)(2^b - 1)) \\
&= (2^a - 1, 2^b + 1)(2^a - 1, 2^b - 1) && \text{(by (2.15) and (2.18))} \\
&= (2^a - 1, 2^b + 1)(2^{(a,b)} - 1)
\end{aligned}$$

Divide by $2^{(a,b)}-1$ and get

$$\frac{2^{(a,2b)}-1}{2^{(a,b)}-1} = (2^a-1, 2^b+1). \qquad \square$$

Recalling that $\omega = 2^{2r}$, $n = 2^m$ and $K = r2^m$ we can now prove (2.10) by showing that $(\omega^j - 1, 2^K + 1) = 1$, for $j \in [1 : n-1]$. Thus all $\omega^j - 1$ are units and therefore no zero divisors.

$$\begin{aligned}
(\omega^j - 1, 2^K + 1) &= ((2^{2r})^j - 1, 2^{r2^m} + 1) \\
&= (2^{2rj} - 1, 2^{r2^m} + 1) \\
&= \frac{2^{(2rj,2r2^m)} - 1}{2^{(2rj,r2^m)} - 1} && \text{(by (2.19))} \\
&= \frac{2^{2r(j,2^m)} - 1}{2^{2r(j,2^{m-1})} - 1}. && \text{(by (2.14))}
\end{aligned}$$

Since $j < 2^m$ it is clear that $(j, 2^m) = (j, 2^{m-1})$. Hence

$$(\omega^j - 1, 2^K + 1) = \frac{2^{2r(j,2^{m-1})} - 1}{2^{2r(j,2^{m-1})} - 1} = 1. \qquad \square$$

Still open is (2.11). For $n = 2^m$ to be a unit in $\mathbb{Z}/(2^K + 1)\mathbb{Z}$, there must exist an i with $2^m i \equiv 1 \equiv 2^{2K}$. Obviously, $i = 2^{2K-m}$ works. $\square$

2.9.2 Convolutions

Schönhage-Strassen multiplication always computes results modulo $2^N + 1$. If it is used recursively to compute the pointwise products this comes in handy, since it allows multiplications where the results are in $[0 : 2^K]$ without performing a modular reduction. This lowers the FFT length and thus the execution time by a factor of two. We will now see how to use convolutions to accomplish this.

If $a(x) = \sum_{i=0}^{n-1} a_i x^i$ and $b(x) = \sum_{j=0}^{n-1} b_j x^j$ are two polynomials with coefficients a_i, $b_j \in R$, then the coefficients of their product $c(x) := a(x)b(x) = \sum_{k=0}^{2n-1} c_k x^k$ are given by the *(acyclic) convolution formula*

$$c_k = \sum_{i+j=k} a_i b_j. \tag{2.20}$$

Figure 8 shows the product of two polynomials a and b both with degree three. The lines from top-left to bottom-right are convolutions with the dots being products of the individual coefficients. For each convolution the sum of the indices of the coefficient products is constant. As Gilbert Strang put it: "You smell a convolution when [the indices] add to [k]" [Str01].

In the process of the FFT as laid out in Section 2.7, two input polynomials are evaluated at ω^i, $i \in [0 : n-1]$, where ω is a primitive n-th root of unity. Afterwards, the sample values are multiplied pointwise and transformed backwards to get the product polynomial.

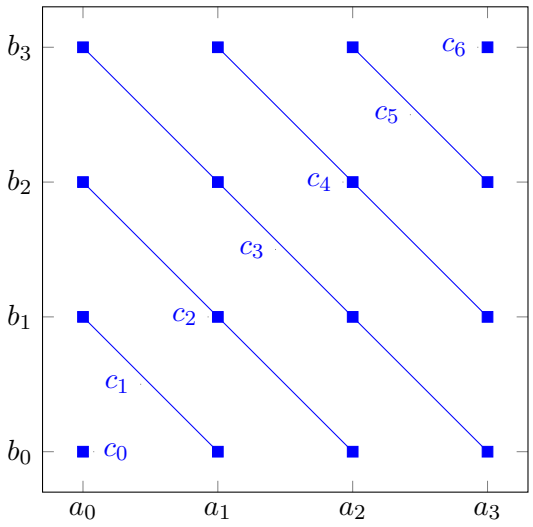

FIGURE 8: Convolution of two polynomials

Define the mapping

$$\phi : R[x] \to R^n,$$
$$a(x) \mapsto (a(\omega^0, \ldots, a(\omega^{n-1})).$$

The kernel of ϕ is the ideal generated by $\prod_{i=0}^{n-1}(x - \omega^i)$. Since $\omega^n = 1$, surely $(\omega^i)^n = 1$ holds as well, for $i \in [0 : n-1]$. So the polynomial $x^n - 1$ yields zero for each $x = \omega^i$, hence it has n distinct roots and the n linear factors $x - \omega^i$. From that we conclude that $\prod_{i=0}^{n-1}(x - \omega^i) = x^n - 1$ and hence that the kernel of ϕ is the ideal generated by $x^n - 1$.

This means that polynomial multiplication that uses the mapping ϕ always gives results modulo $x^n - 1$. This is called the *cyclic* convolution of two polynomials. Given the aforementioned polynomials $a(x)$ and $b(x)$ it produces the product polynomial $c(x)$ with coefficients

$$c_k = \sum_{\substack{i+j \equiv k \\ (\text{mod } n)}} a_i b_j. \tag{2.21}$$

Figure 9 shows the cyclic convolution of two polynomials of degree three. Here, the upper half of coefficients "wraps around" and is added to the lower half. This is why it is sometimes called a *wrapped* convolution.

We now know that a cyclic convolution gives us results modulo $x^n - 1$. Can we get results modulo $x^n + 1$? Schönhage shows us we can.

Since i, j, $k < n$ we can write (2.21) as

$$c_k = \sum_{i+j=k} a_i b_j \; + \sum_{i+j=n+k} a_i b_j. \tag{2.22}$$

The second sum contains the higher half product coefficients that wrap around and are *added* to the lower half coefficients, since $x^n \equiv 1$. But if we want results modulo $x^n + 1$, it holds that

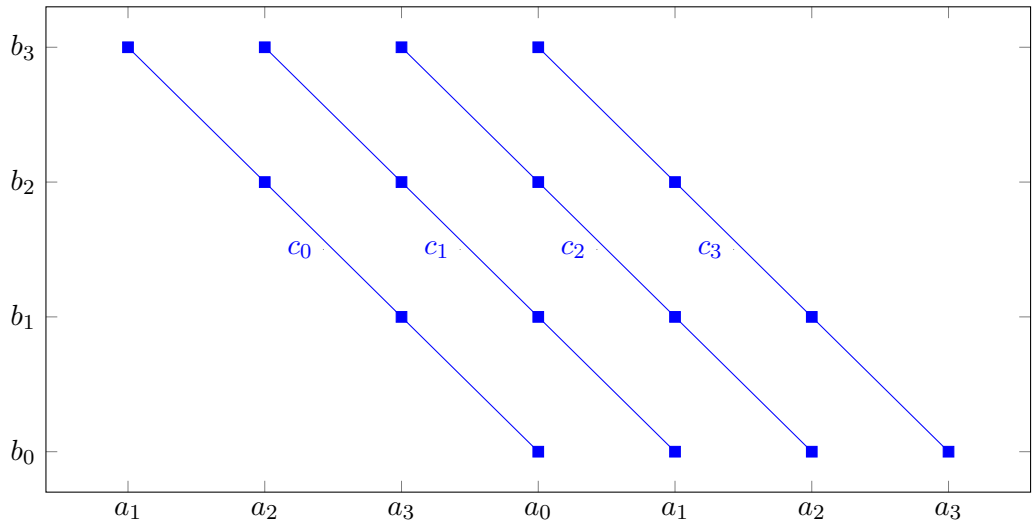

FIGURE 9: Cyclic convolution of two polynomials

$x^n \equiv -1$, hence what we are looking for is a way to compute

$$c_k = \sum_{i+j=k} a_i b_j \; - \sum_{i+j=n+k} a_i b_j. \tag{2.23}$$

Schönhage's idea is to weight each of the coefficients a_i and b_j prior to the cyclic convolution in such a way that for $i + j = n + k$ and $k < n$ it holds that $\theta^n a_i b_j = -a_i b_j$, for some $\theta \in R$ that we will specify immediately. This puts the desired minus sign in front of the second term in (2.23).

Choose the weight θ as follows: let θ be a primitive n-th root of -1, that is, $\theta^n = -1$ and hence $\theta^2 = \omega$. To compute (2.23), we use (2.21), but weight the inputs like

$$\widetilde{a}_i := \theta^i a_i \quad \text{and} \quad \widetilde{b}_j := \theta^j b_j \tag{2.24}$$

and apply the proper "counterweight" θ^{-k} to the whole sum, so we get

$$\begin{aligned} c_k &= \theta^{-k} \sum_{\substack{i+j \equiv k \\ (\text{mod } n)}} \widetilde{a}_i \widetilde{b}_j \qquad (2.25) \\ &= \theta^{-k} \Big(\sum_{i+j=k} \widetilde{a}_i \widetilde{b}_j \; + \sum_{i+j=n+k} \widetilde{a}_i \widetilde{b}_j \Big) \\ &= \theta^{-k} \sum_{i+j=k} \theta^i a_i \theta^j b_j \; + \; \theta^{-k} \sum_{i+j=n+k} \theta^i a_i \theta^j b_j \\ &= \theta^{-k} \sum_{i+j=k} \theta^k a_i b_j \; + \; \theta^{-k} \sum_{i+j=n+k} \theta^{n+k} a_i b_j \\ &= \sum_{i+j=k} a_i b_j \; + \; \theta^n \sum_{i+j=n+k} a_i b_j \\ &= \sum_{i+j=k} a_i b_j \; - \sum_{i+j=n+k} a_i b_j. \end{aligned}$$

□

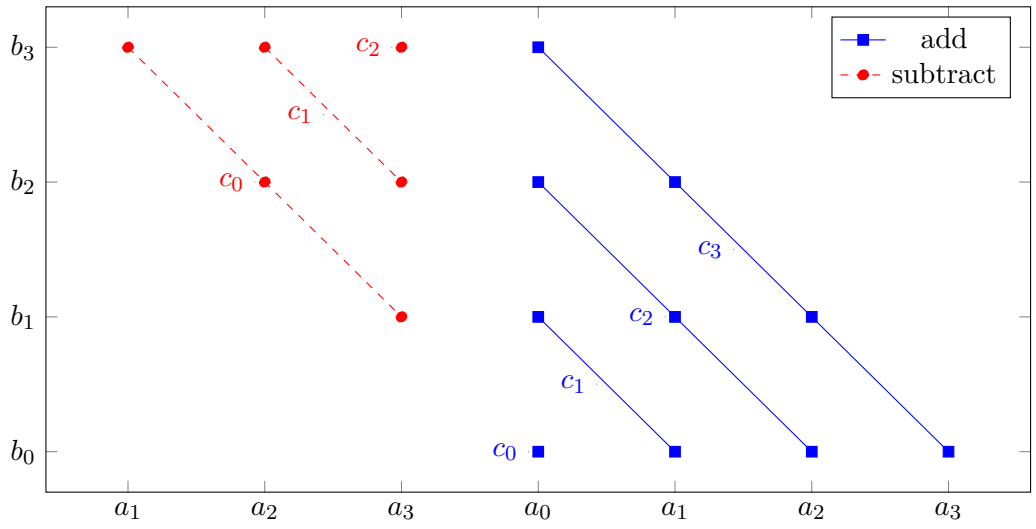

FIGURE 10: Negacyclic convolution of two polynomials

This is called a *negacyclic* or *negative wrapped* convolution. Figure 10 shows a diagram of it. Please note that θ is in $\mathbb{Z}/(2^K+1)\mathbb{Z}$ as well a power of 2, so weighting can be done by a cyclic shift.

According to (2.23), the c_k can become negative. Yet, we are looking for nonnegative $c'_k \equiv c_k \pmod{2^K+1}$ with $c'_k \in [0 : 2^K]$. If $c_k < 0$, we can find $c'_k := c_k + 2^K + 1$.

2.9.3 The Procedure

We are now all set to describe the whole procedure: given nonnegative integers a and b find their product $c := ab$ modulo $2^N + 1$.

Since the product is computed modulo 2^N+1, we must choose N big enough for the full product c. If we choose $N \geq \lceil \log a \rceil + \lceil \log b \rceil$ this is surely the case.

Denote R the ring $\mathbb{Z}/(2^K+1)\mathbb{Z}$, for some $K = r2^m$. Let $n := 2^m$ be the FFT length and let $s := \lceil N/n \rceil$ be the bit length of input coefficients cut from a and b. Then our choice of parameters has to meet the following constraints:

- R must contain a primitive n-th root of unity ω that is an even power of 2. $(2^{2x})^n \equiv 1 \equiv 2^{2K}$ leads to the sufficient condition
$$n \mid K. \tag{2.26}$$
- R must be big enough to hold the convolution sums. Because of (2.23), the $c_k \in [-n2^{2s}+1 : n2^{2s}-1]$, so the total range has size $2n2^{2s}-1$. Hence select K so that $2^K+1 > 2n2^{2s}-1 = 2^{m+2s+1}-1$. It is sufficient to select
$$K \geq m + 2s + 1. \tag{2.27}$$
- For increased speed, we might want to choose a larger K that contains a higher power of 2. We will perform benchmarking later to find out if it pays off.

These constraints lead to values for the FFT length $n := 2^m$, the number of input bits per coefficient $s := \lceil N/n \rceil$, and $K = r2^m \geq m + 2s + 1$. This in turn forces a new, maybe slightly higher value for $N := s2^m$, and determines $\omega := 2^{2r}$ and $\theta := 2^r$. Given those parameters, we can proceed like we did with `QMUL` in Section 2.8, but with some alterations:

1. Split both input numbers a and b into n coefficients of s bits each. Use at least $K+1$ bits to store them, to allow encoding of the value 2^K.
2. Weight both coefficient vectors according to (2.24) with powers of θ by performing cyclic shifts on them.
3. Shuffle the coefficients a_i and b_j.
4. Evaluate a_i and b_j. Multiplications by powers of ω are cyclic shifts.
5. Do n pointwise multiplications $c_k := a_k b_k$ in $\mathbb{Z}/(2^K + 1)\mathbb{Z}$. If `SMUL` is used recursively, provide K as parameter. Otherwise, use some other multiplication function like `T3MUL` and reduce modulo $2^K + 1$ afterwards.
6. Shuffle the product coefficients c_k.
7. Evaluate the product coefficients c_k.
8. Apply the counterweights to the c_k according to (2.25). Since $\theta^{2n} \equiv 1$ it follows that $\theta^{-k} \equiv \theta^{2n-k}$.
9. Normalize the c_k with $1/n \equiv 2^{-m}$ (again a cyclic shift).
10. Add up the c_k and propagate the carries. Make sure to properly handle negative coefficients.
11. Do a reduction modulo $2^N + 1$.

If `SMUL` is used recursively, its input parameter N cannot be chosen freely. The calling `SMUL` provides its parameter K as the input parameter N of the called `SMUL`.

I implemented some optimizations to the procedure outlined above to save execution time:

- Steps 1, 2 and 3 can be combined. Furthermore, since some high part of a and b is virtually zero-padded, initialization of that part can be done quickly.
- Steps 8 and 9 can be combined.
- On the outermost `SMUL`, where N can be chosen, we don't need to do a negacyclic transform. This lets us skip the weighting of a_i, b_j and c_k in Steps 2 and 8. We don't check for negative coefficients in Step 10 and don't need the reduction in Step 11. Furthermore, we don't need $\theta = \sqrt{\omega}$ and thus can extend the FFT length by another factor of 2. The sufficient condition for selecting K relaxes to $n \mid 2K$.
- The cyclic shifts often shift by multiples of the word size w, where a word-sized copy is faster than access to individual bits.

2.9.4 Run-time Analysis

Let us analyze the cost of `SMUL` to compute a product modulo $2^N + 1$ and call this cost $T(N)$.

According to (2.26), it is clear that $K \geq 2^m$, but we will show the time bound for the condition

$$K = 2^m. \tag{2.28}$$

This means that we might have to choose a K that is larger than required by (2.26) and (2.27), but our choice only increases K by at most a factor of 2.

Furthermore, according to (2.27), $K \geq m + 2s + 1$, where $s = \lceil N/2^m \rceil$. Surely we will find a suitable $K = 2^m \leq 2(m + 2s + 1)$. So for sufficiently large values of N

$$\begin{aligned} m + 2N/2^m + 1 \leq K &\leq 2m + 4N/2^m + 2 \\ 2N/K \leq K &\leq 5N/K \\ 2N \leq K^2 &\leq 5N. \qquad (2.29) \\ \sqrt{2N} \leq K &\leq \sqrt{5N}. \qquad (2.30) \end{aligned}$$

Steps 1, 3, 6 and 10 have obviously cost $O(2^m K)$. The same applies to Steps 2, 8 and 9, since the cost of cyclic shifts modulo $2^K + 1$ is $O(K)$ as well. By the same argument Step 11 has cost $O(N)$.

According to (2.4), the FFT evaluation costs $O(n \log n)$, with $n = 2^m$, but we have to take into account that in contrast to (2.4), multiplications by roots of unity don't cost $O(1)$ here, but $O(K)$, so the cost of evaluation in Steps 4 and 7 is $O(m2^m)O(K)$. That leaves Step 5, where we have 2^m multiplications modulo $2^K + 1$, so the cost of that is $2^m T(K)$.

If we add everything up, we get for the total cost

$$T(N) = O(2^m K) + O(N) + O(m2^m)O(K) + 2^m T(K).$$

Using (2.28) and (2.29) we get

$$\begin{aligned} T(N) &= O(N) + O(mN) + KT(K) \\ &= O(mN) + KT(K). \end{aligned}$$

By (2.30) we know that $2^m = K \leq \sqrt{5N}$, hence $m \leq \frac{1}{2} \log (5N)$. Ergo

$$T(N) = O(N \log N) + O(\sqrt{N})T(\sqrt{5N}).$$

Unrolling the recursion once leads to

$$\begin{aligned} T(N) &= O(N \log N) + O(\sqrt{N})(O(\sqrt{5N} \log \sqrt{5N}) + O(\sqrt[4]{5N})T(\sqrt[4]{5^3 N})) \\ &= O(N \log N) + O(\sqrt{N})(O(\sqrt{N} \log N) + O(\sqrt[4]{N})T(\sqrt[4]{5^3 N})) \\ &= O(N \log N) + O(N \log N) + \underbrace{O(\sqrt[4]{N^3})T(\sqrt[4]{5^3 N})}_{=:\Delta}. \end{aligned}$$

After $\log\log N$ steps the remaining summand $\Delta \leq O(N)$:

$$\begin{aligned} T(N) &= O(N\log N) + \underbrace{O(N\log N) + \ldots}_{\log\log N \text{ times}} + O(N) \\ &= O(N\log N) + O(N\log N)\log\log N + O(N) \\ &= O(N \cdot \log N \cdot \log\log N). \end{aligned} \tag{2.31}$$

To see why it takes $\log\log N$ steps, observe that the order of the root doubles with each recursion step. Hence, after $\log\log N$ steps the order has reached $2^{\log\log N} = \log N =: \lambda$. So the remaining summand $\Delta \leq O(\sqrt[\lambda]{N^{\lambda-1}})T(\sqrt[\lambda]{5^{\lambda-1}N}) \leq O(N)T(5\sqrt[\lambda]{N})$. Lastly, $\sqrt[\lambda]{N} = N^{1/\log N} = 2$ and hence $\Delta \leq O(N)T(10) \leq O(N)$.

Until the discovery of Fürer's algorithm [Fü07] in 2007 this was the lowest known time bound for a multiplication algorithm.

Now let us look at memory requirements. Memory needed for all 2^m coefficients of one input number is $2^m K$ bits. According to (2.27) with $s = \lceil N/2^m \rceil$ it holds that $K \geq m + 2\lceil N/2^m \rceil + 1$. Hence memory requirements in bits for one polynomial are

$$\begin{aligned} 2^m K &\geq 2^m \cdot (m + 2\lceil N/2^m \rceil + 1) \\ &\geq 2^m \cdot (m + 2N/2^m + 1) \\ &\geq 2^m \cdot 2N/2^m \\ &\geq 2N. \end{aligned}$$

Temporary memory is required for both input polynomials, but for the resulting polynomial storage of one of the input polynomials can be reused. `SMUL` needs some memory for the multiplication of sample points, but this is only of the size of one coefficient, that is, K bits and doesn't change the order of the approximation. Hence, if N denotes the bit length of the product and $M_{\texttt{SMUL}}(N)$ denotes total memory required by `SMUL`, it holds that

$$M_{\texttt{SMUL}}(N) \approx 4N \text{ bits}. \tag{2.32}$$

Figure 20 shows measured memory requirements, but note that in that table N refers to the bit length of one *input*, where in this section N denotes the bit length of the *product*.

2.9.5 Benchmarking

Apart from optimizing the implementation on higher (see page 28) and lower levels (like assembly language subroutines) benchmarking shows that we can save quite some execution time by finding the fastest FFT length n from all possible values.

For this, we measure execution cycles for multiplications with different possible FFT lengths. In principle, larger FFT lengths lead to faster multiplications, but the largest possible FFT length is usually not the fastest. Larger FFT lengths lead to smaller coefficient sizes, but more operations on the coefficients. On the other hand, the value of the primitive n-th root ω might allow byte- or even word aligned (or even better SSE-word aligned) shifts, which can be implemented faster than general bit-shifts. The smaller the FFT length, the better the alignment for the cyclic shifts.

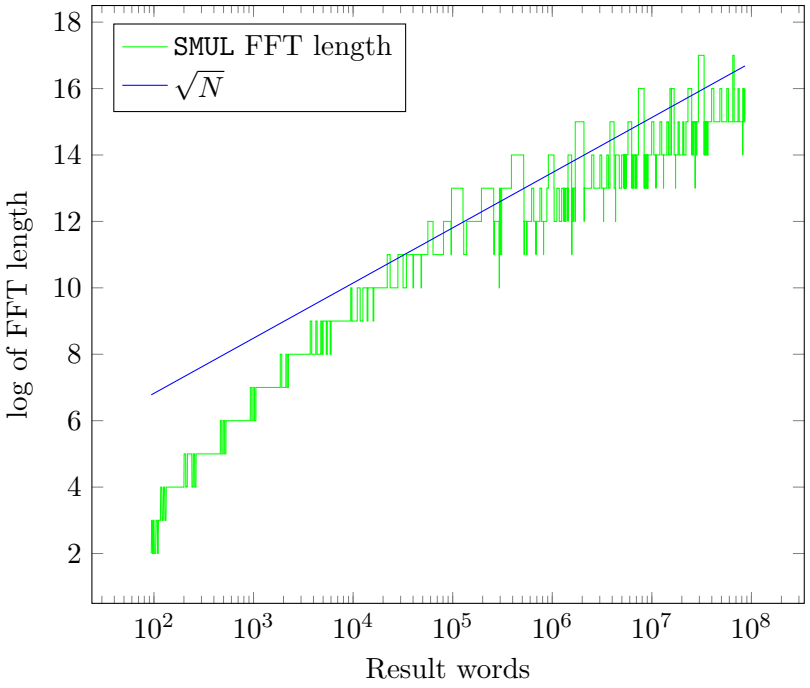

FIGURE 11: SMUL FFT length vs. input length

Maybe even more importantly, certain values of K that contain high powers of 2 allow for larger FFT lengths in the recursively called SMUL. So sometimes larger K work much faster, even if the FFT length stays unchanged.

As a result, the fastest FFT length switches several times until it settles for a higher value. Figure 11 gives an impression of this. The $\sqrt{N}$ graph is printed for orientation, since $2^m \geq \sqrt{2N/r}$.

The graph of execution cycles vs. input lengths is shown in Figure 12. We can see that it is well below the QMUL graph, but intersects with the T3MUL graph at about 2500 words, that is, about 47 500 decimal digits. Furthermore, we observe a certain "bumpiness" of the graph, which is a result of the changing FFT lengths and ring sizes. Yet, it is much smoother than the QMUL graph.

Lastly, we try to model the run-time according to its theoretical value (2.31) for large values of N. If we write the run-time with an explicit constant, then

$$T_\sigma(N) \leq \sigma \cdot N \cdot \log N \cdot \log\log N. \tag{2.33}$$

Dividing measured execution cycles by $N \cdot \log N \cdot \log\log N$ to calculate σ leads to the graph depicted in Figure 13. Please note that here N is the length of the product in *bits*. Interestingly, this graph seems to have two plateau-like sections.

The first plateau ranges roughly from 12 800 to 8 000 000 input bits and the second plateau starts at about 32 000 000 input bits. Since SMUL requires about $4N$ bits of temporary memory, the above numbers indicate a plateau from 12 KB to 8 MB and another starting from 32 MB temporary memory. This corresponds quite nicely with the cache sizes of the test machine (see Appendix A). Such an influence on the run-time constant σ is no longer visible only after the

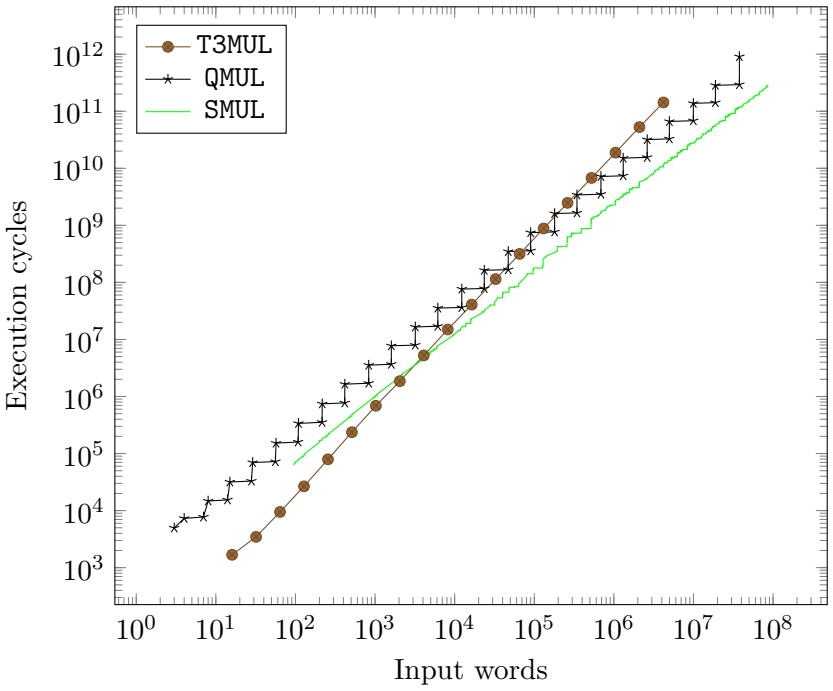

FIGURE 12: Execution time of `SMUL`

required temporary memory is some orders of magnitude larger than the cache size. In our case that would be starting from about 32 MB temporary memory.

Since after level 3 there are no other caching mechanisms, we can average σ for input sizes above 32 Mbits (since a word is 64 bits, this equals 512 Kwords) and that leads to an average $\sigma \approx 0.3159$.

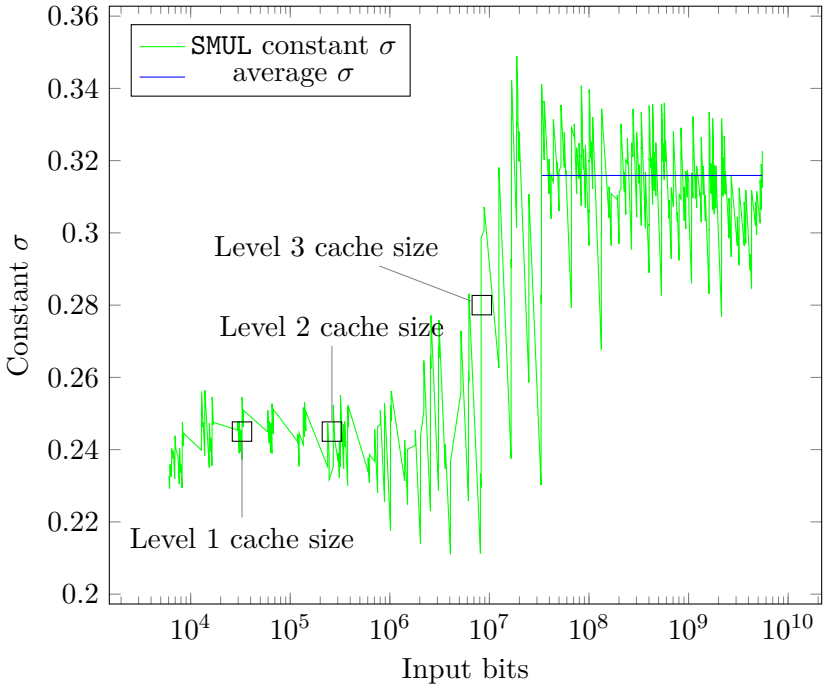

FIGURE 13: `SMUL` run-time constant σ

2.9.6 Outlook

There are some possible improvements which might lower execution time that I have so far not implemented or tested. Namely, this is:

- I benchmark different FFT lengths n to find the fastest one, but I could benchmark different ring sizes K as well. It is sometimes profitable to use larger values for K, if K contains higher powers of 2.
- Schönhage's $\sqrt{2}$ knack[†][SGV94, p. 36, exercise 18]. We can increase the transform length by a factor of 2 by noticing that $\xi = 2^{3K/4} - 2^{K/4}$ is a primitive $4K$-th root of unity, since $\xi^2 \equiv 2 \pmod{2^K+1}$ and $2^{2K} \equiv 1 \pmod{2^K+1}$. In [GKZ07], the authors mention a 10 % speed increase. That paper also contains some other promising fields of optimization.
- The current implementation cuts input numbers into coefficients only at word boundaries. Maybe cutting them at bit boundaries might lower K slightly for some input lengths.
- The procedure outlined in [CP05, pp. 502–503] saves one bit in K by selecting $K \geq m + 2s$ and then uses a sharper bound for the c_k than I did by noticing that each c_k can only have $k+1$ positively added summands, see Figure 10.

[†][GKZ07] call it $\sqrt{2}$ *trick*, but Schönhage interjects that "trick" sounds too much like a swindle, so I call it a *knack* instead.

Chapter 3

The DKSS Algorithm

This chapter describes DKSS multiplication, especially how it employs the fast Fourier transform, and analyzes its execution time theoretically. Finally, differences between my implementation and the DKSS paper are described.

3.1 Overview

Schönhage and Strassen's algorithm for fast multiplication of large numbers (implemented as `SMUL`, see Section 2.9) uses the ring $R = \mathbb{Z}/(2^K + 1)\mathbb{Z}$ and exploits the fact that 2 is a primitive $2K$-th root of unity in R. This permits the crucial speed-up in the fast Fourier transform: multiplications by powers of the root of unity can be realized as cyclic shifts and are thus considerably cheaper. An N-bit number is broken down into numbers that are $O(\sqrt{N})$ bits long and when sample values are multiplied, the same algorithm is used recursively.

The DKSS algorithm (its implementation is called `DKSS_MUL` here) keeps this structure, but extends it further. Where `SMUL` used the ring $\mathbb{Z}/(2^K + 1)\mathbb{Z}$ with 2 as root of unity, DKSS multiplication uses the polynomial quotient ring $\mathcal{R} := \mathcal{P}[\alpha]/(\alpha^m + 1)$. Since $\alpha^m \equiv -1$, α is a primitive $2m$-th root of unity and again multiplications by powers of the root of unity can be done as cyclic shifts. Underlying $\mathcal{R}$ is the ring $\mathcal{P} := \mathbb{Z}/p^z\mathbb{Z}$, where p is a prime number and z is a constant. This "double structure" can be exploited in the FFT and allows to break down an N-bit input number into numbers of $O(\log^2 N)$ bits.

In their paper [DKSS13], De, Kurur, Saha and Saptharishi describe the algorithm without any assumption about the underlying hardware. Since we are interested in an actual implementation, we can allow ourselves some simplifications, namely the precomputation of the prime p, and as a consequence drop their concept of k-variate polynomials by setting $k = 1$. Section 3.4 explains the differences between my implementation and the original paper in more detail.

3.2 Formal Description

We want to multiply two nonnegative integers a, $b < 2^N$, $N \in \mathbb{N}$ to obtain their product $c := ab < 2^{2N}$. As usual, we convert the numbers into polynomials over a ring (denoted $\mathcal{R}$), use the fast Fourier transform to transform their coefficients, then multiply the sample values

and transform backwards to gain the product polynomial. From there, we can easily recover the resulting integer product.

Denote $\mathcal{R} \coloneqq \mathcal{P}[\alpha]/(\alpha^m + 1)$. As usual, we identify $\mathcal{R}$ with the set of all polynomials in $\mathcal{P}[\alpha]$ which are of degree less than m and where polynomial multiplication is done modulo $(\alpha^m + 1)$. Polynomial coefficients are in $\mathcal{P}$ and are called *inner* coefficients. Furthermore, define $\mathcal{P} \coloneqq \mathbb{Z}/p^z\mathbb{Z}$, where p is a prime number and z is a constant chosen independently of the input. We will see how to choose p shortly.

Input numbers a and b are encoded as polynomials $a(x)$ and $b(x) \in \mathcal{R}[x]$ with degree-bound M. That is, $a(x)$ and $b(x)$ are polynomials over $\mathcal{R}$ whose coefficients are themselves polynomials over $\mathcal{P}$. Call the coefficients of $a(x)$ and $b(x)$ *outer* coefficients.

This outline shows how to multiply a and b. The following Sections 3.2.1 – 3.2.8 contain the details.

1. Choose integers $m \geq 2$ and $M \geq m$ as powers of 2, such that $m \approx \log N$ and $M \approx N/\log^2 N$. We will later perform FFTs with length $2M$, while m is the degree-bound of elements of $\mathcal{R}$. For simplicity of notation, define $\mu \coloneqq 2M/2m$.
2. Let $u \coloneqq \lceil 2N/Mm \rceil$ denote the number of input bits per inner coefficient. Find a prime p with $2M \mid p - 1$ and $p^z \geq Mm2^{2u}$. The prime power p^z is the modulus of the elements of $\mathcal{P}$.
3. From parameters M, m and p compute a principal (see Section 3.2.2 for definition) $2M$-th root of unity[†] $\rho \in \mathcal{R}$ with the additional property that $\rho^{2M/2m} = \alpha$. This property plays an important part in Step 5.
4. Encode a and b as polynomials $a(x)$, $b(x) \in \mathcal{R}[x]$ with degree-bound M. To accomplish that, break them into M blocks with $um/2$ bits in each block. Each such block describes an outer coefficient. Furthermore, split those blocks into $m/2$ blocks of u bits each, where each block forms an inner coefficient in the lower-degree half of a polynomial. Set the upper $m/2$ inner coefficients to zero. Finally, set the upper M outer coefficients to zero to stretch $a(x)$ and $b(x)$ to degree-bound $2M$.
5. Use root ρ to perform a length-$2M$ fast Fourier transform of $a(x)$ and $b(x)$ to gain $\widehat{a}_i \coloneqq a(\rho^i) \in \mathcal{R}$, likewise $\widehat{b}_i$. Use the special structure of $\mathcal{R}$ to speed up the FFT.
6. Multiply components $\widehat{c}_i \coloneqq \widehat{a}_i\widehat{b}_i$. Note that $\widehat{a}_i$, $\widehat{b}_i \in \mathcal{R}$ are themselves polynomials. Their multiplication is reduced to integer multiplication and the DKSS algorithm is used recursively.
7. Perform a backwards transform of length $2M$ to gain the product polynomial $c(x) \coloneqq a(x)b(x)$.
8. Evaluate the inner polynomials of the product polynomial $c(x)$ at $\alpha = 2^u$ and the outer polynomials at $x = 2^{um/2}$ to recover the integer result $c = ab$.

3.2.1 Choosing M and m

Choose $m \geq 2$ and $M \geq m$ as powers of 2, such that $m \approx \log N$ and $M \approx N/\log^2 N$. For the run-time analysis, the bounds $M = O(N/\log^2 N)$ and $m = O(\log N)$ are more convenient.

[†]We simply write ρ instead of $\rho(\alpha)$, keeping in mind that ρ itself is a polynomial in α.

3.2.2 Finding the Prime p

We use the following definition that captures the requirements for the existence of the inverse FFT transform (cf. Sections 2.7 and 2.9.1):

Definition. *Let R be a commutative ring with unity. A primitive n-th root of unity $\zeta \in R$ is called* principal *if and only if $\sum_{i=0}^{n-1}(\zeta^j)^i = 0$, for $j \in [1 : n-1]$, and n is coprime to the characteristic of R.*

Since numbers are encoded as polynomials with degree-bound M (Step 4) and then multiplied, the result has a degree-bound of $2M$, so we need a principal $2M$-th root of unity for the FFTs. If $p := h \cdot 2M + 1$ is prime for some $h \in \mathbb{N}$ (primes of this form are called *Proth primes*), we can compute a principal $2M$-th root of unity ω in $\mathbb{Z}/p^z\mathbb{Z}$. Section 3.2.3 shows how it is done.

Why is $p^z \geq Mm2^{2u}$ required? Since both $a(x)$ and $b(x)$ have degree-bound M, each outer coefficient of their product $c(x)$ is the sum of up to M outer coefficient products. Each of these products is the sum of up to $m/2$ inner coefficient products, with each factor $< 2^u$ by construction. So the inner coefficients can take values as high as $\frac{1}{2}Mm(2^u-1)^2$. If we choose $p^z \geq Mm2^{2u}$, we are on the safe side.

But does a prime of the form $p = h \cdot 2M + 1$ exist for all M? We can answer that in the affirmative with the help of the following

Theorem (Linnik [Lin44a], [Lin44b])**.** *For any pair of coprime positive integers d and n, the least prime p with $p \equiv d \pmod{n}$ is less than ℓn^L, where ℓ and L are positive constants.*[†]

We want to show the existence of a prime p with $p \equiv 1 \pmod{2M}$, but also require $p^z \geq Mm2^{2u}$. Since Linnik's Theorem makes only a statement about the *first* prime, we must check that this prime to a constant power matches the requirement. An easy calculation shows that $(2M+1)^6 \geq Mm2^{2u}$. As p is of the form $p = h \cdot 2M + 1$, we see that for every $h \in \mathbb{N}$ and every $z \geq 6$ this means that $p^z \geq Mm2^{2u}$. With the size condition resolved, we use Linnik's theorem to show that $p < \ell(2M)^L$. □

To get an estimate of p in terms of N, we recall that $M = O(N/\log^2 N)$ and see that

$$p < \ell(2M)^L = O\Big(\Big(\frac{N}{\log^2 N}\Big)^L\Big) = O\Big(\frac{N^L}{\log^{2L} N}\Big). \tag{3.1}$$

In the implementation I tested candidate primes p for primality by using the Lucas-test [CP05, sec. 4.1] that allows for fast deterministic primality testing if the full factorization of $p-1$ is known. $p-1$ is a power of 2 times a small factor, because $p = h \cdot 2M + 1$, so this test is well suited here.

With that in mind, we can precompute values for p for all possible lengths N, since our supported hardware is 64-bit and hence $N < 2^{67}$ and (assuming $M \approx N/\log^2 N$) $M < 2^{55}$.

3.2.3 Computing the Root of Unity ρ

In Step 2 we computed a prime $p = h \cdot 2M + 1$, $h \in \mathbb{N}$. Now we want to find a $2M$-th root of unity ω in $\mathbb{Z}/p^z\mathbb{Z}$. A generator ζ of $\mathbb{F}_p^* = \{1, 2, \ldots, p-1\}$ has order $p - 1 = h \cdot 2M$. Hence ζ

[†]Over the years, progress has been made in determining the size of *Linnik's constant* L. A recent work by Xylouris [Xyl11] shows that $L \leq 5$.

is a primitive $(p-1)$-th root of unity and ζ^h a primitive $2M$-th root of unity in $\mathbb{Z}/p\mathbb{Z}$. In fact, both ζ and ζ^h are even principal. The following theorem allows us to find roots in $\mathbb{Z}/p^s\mathbb{Z}$ for integer values $s \geq 2$:

Theorem (Hensel Lifting [NZM91, sec. 2.6]). *Let $f \in \mathbb{Z}[x]$ and let $\zeta_s \in \mathbb{Z}$ be a solution to $f(x) \equiv 0 \pmod{p^s}$, such that $f'(\zeta_s)$ is a unit in $\mathbb{Z}/p\mathbb{Z}$. Then $\zeta_{s+1} := \zeta_s - f(\zeta_s)/f'(\zeta_s)$ solves $f(x) \equiv 0 \pmod{p^{s+1}}$ and furthermore $\zeta_{s+1} \equiv \zeta_s \pmod{p^s}$.*

Finding a primitive $(p-1)$-th root of unity in $\mathbb{Z}/p^z\mathbb{Z}$ means solving $f(x) = x^{p-1} - 1$. We can use Hensel lifting, because $f'(\zeta_s) = (p-1)\zeta_s^{p-2}$ is a unit in $\mathbb{Z}/p\mathbb{Z}$, since $p - 1 \neq 0$ and $\zeta_s^{p-2} \equiv \zeta^{p-2} \not\equiv 0 \pmod{p}$. If we start with $x = \zeta$ as solution to $f(x) \equiv 0 \pmod{p}$, then repeated lifting yields a $(p-1)$-th root of unity ζ_z in $\mathbb{Z}/p^z\mathbb{Z}$. Hence $\omega := \zeta_z^h$ is a $2M$-th root of unity in $\mathbb{Z}/p^z\mathbb{Z}$. To see that ω is also primitive, let $j \in [1 : 2M-1]$. Then $\omega^j = \zeta_z^{hj} \equiv \zeta^{hj} \not\equiv 1 \pmod{p}$, as ζ is a primitive $(p-1)$-th root of unity in $\mathbb{Z}/p\mathbb{Z}$.

To prove that ω is even principal note that the characteristic of $\mathcal{R}$ is p^z, so ω has to be coprime to p^z, that is, coprime to p. But $\omega = \zeta_z^h \equiv \zeta^h \not\equiv 0 \pmod{p}$, so ω is not a multiple of p. Hence ω and p^z are coprime. Furthermore, it holds for $j \in [1 : 2M-1]$ that

$$\sum_{i=0}^{2M-1} (\omega^j)^i = \frac{1-\omega^{j2M}}{1-\omega^j} = \frac{1-(\omega^{2M})^j}{1-\omega^j} \equiv 0 \pmod{p^z},$$

because ω is a primitive $2M$-th root of unity in $\mathbb{Z}/p^z\mathbb{Z}$. □

We are looking for a principal $2M$-th root of unity $\rho \in \mathcal{R}$ with the additional property $\rho^{2M/2m} = \alpha$. Since $\mathcal{R} = \mathcal{P}[\alpha]/(\alpha^m + 1)$, α is a principal $2m$-th root of unity. Denote $\gamma := \omega^{2M/2m}$, a principal $2m$-th root of unity in $\mathcal{P}$. Observe that γ^i is a root of $\alpha^m + 1 = 0$, for an odd $i \in [1 : 2m-1]$, since $(\gamma^i)^m = (\gamma^m)^i = (-1)^i = -1$. Because the γ^i are pairwise different it follows that

$$\prod_{\substack{i=1 \\ i \text{ odd}}}^{2m-1} (\alpha - \gamma^i) = \alpha^m + 1.$$

Theorem (Chinese Remainder Theorem [Fis11, sec. 2.11]). *If R is a commutative ring with unity and $I_1, \ldots, I_k$ are ideals of R, which are pairwise coprime (that is, $I_i + I_j = R$, for $i \neq j$), then the mapping*

$$\begin{aligned} \phi : R &\to R/I_1 \times \ldots \times R/I_k, \\ x &\mapsto (x + I_1, \ldots, x + I_k) \end{aligned}$$

is surjective and $\ker \phi = I_1 \cdot \ldots \cdot I_k$. Especially, $\phi(x) = \phi(x') \Leftrightarrow x - x' \in I_1 \cdot \ldots \cdot I_k$ and

$$R/(I_1 \cdot \ldots \cdot I_k) \cong R/I_1 \times \ldots \times R/I_k.$$

If the ideals I_i are generated by $(\alpha - \gamma^i)$, they are pairwise coprime, since $\gamma^i - \gamma^j$ is a unit in $\mathcal{R}$, for $i \neq j$, see (3.3) below. So $\alpha \in \mathcal{P}[\alpha]/(\alpha^m + 1)$ is isomorphic to the k-tuple of remainders $(\gamma, \gamma^3, \ldots, \gamma^{2m-1}) \in \prod_i (\mathcal{P}[\alpha]/I_i)$. We are looking for a ρ satisfying $\rho^{2M/2m} = \alpha$, but we already know that $\omega^{2M/2m} = \gamma$, hence $(\omega, \omega^3, \ldots, \omega^{2m-1})$ is the tuple of remainders isomorphic to ρ. To regain $\rho \in \mathcal{P}[\alpha]/(\alpha^m + 1)$ we use the next

Theorem (Lagrange Interpolation). *Let R be a commutative ring with unity. Given a set of k data points $\{(x_1, y_1), \ldots, (x_k, y_k)\}$ with $(x_i, y_i) \in R \times R$, where the x_i are pairwise different*

and $x_i - x_j$ is a unit for all $i \neq j$, there exists a polynomial $L(x)$ of degree less than k passing through all k points (x_i, y_i). This polynomial is given by

$$L(x) \coloneqq \sum_{i=1}^{k} y_i \ell_i(x), \qquad \textit{where} \qquad \ell_i(x) \coloneqq \prod_{\substack{j=1 \\ j \neq i}}^{k} \frac{x - x_j}{x_i - x_j}.$$

In our case we know that $\rho(\alpha) \cong (\omega, \omega^3, \ldots, \omega^{2m-1})$, so it follows that the set of data points is $\{(\gamma, \omega), (\gamma^3, \omega^3), \ldots, (\gamma^{2m-1}, \omega^{2m-1})\}$ and hence

$$\rho(\alpha) \coloneqq \sum_{\substack{i=1 \\ i \text{ odd}}}^{2m-1} \omega^i \ell_i(\alpha), \qquad \text{where} \qquad \ell_i(\alpha) \coloneqq \prod_{\substack{j=1 \\ j \neq i \\ j \text{ odd}}}^{2m-1} \frac{\alpha - \gamma^j}{\gamma^i - \gamma^j}. \tag{3.2}$$

The inverses to $\gamma^i - \gamma^j$ exist. To see why, observe that an element of $\mathbb{Z}/p^z\mathbb{Z}$ is a unit if and only if it is not divisible by p. But

$$\begin{aligned} \gamma^i - \gamma^j &= \zeta_z^{i(p-1)/2m} - \zeta_z^{j(p-1)/2m} \\ &\equiv \zeta^{i(p-1)/2m} - \zeta^{j(p-1)/2m} \pmod{p} \\ &\not\equiv 0 \pmod{p}, \end{aligned} \tag{3.3}$$

because ζ is a primitive $(p-1)$-th root of unity and $i, j \in [1 : 2m-1]$ and since $i \neq j$ the two exponents of ζ are different. □

3.2.4 Distribution of Input Bits

We want to encode a nonnegative integer $a < 2^N$ as polynomial over $\mathcal{R}[x]$ with degree-bound M. We already calculated $u = \lceil 2N/Mm \rceil$, the number of bits per inner coefficient. First, a is split into M blocks of $um/2$ bits each, starting at the lowest bit position. Each of these blocks encodes one outer coefficient. Since $Mum/2 \geq N$, we might need to zero-pad a at the top.

Then, each of the M outer coefficient blocks is broken into $m/2$ blocks, each u bits wide. They form the inner coefficients. Since the inner coefficients describe a polynomial with degree-bound m, the upper half of the coefficients is set to zero.

Finally, set the upper M outer coefficients to zero to stretch $a(x)$ to degree-bound $2M$. Figure 14 depicts this process.

3.2.5 Performing the FFT

Section 2.6 described a radix-2 Cooley-Tukey FFT. The DKSS algorithm uses an FFT with a higher radix, but still the same basic concept. A Cooley-Tukey FFT works for any length that is a power of 2, here the length is $2M$ and it can be split as $2M = 2m \cdot \mu$, with $\mu = 2M/2m$.

The DKSS algorithms uses a radix-μ decimation in time Cooley-Tukey FFT (cf. [DV90, sec. 4.1]), that is, it first does μ FFTs of length $2m$, then multiplies the results by "twiddle factors" and finally performs $2m$ FFTs of length μ. We can exploit the fact that the length-$2m$ FFT uses

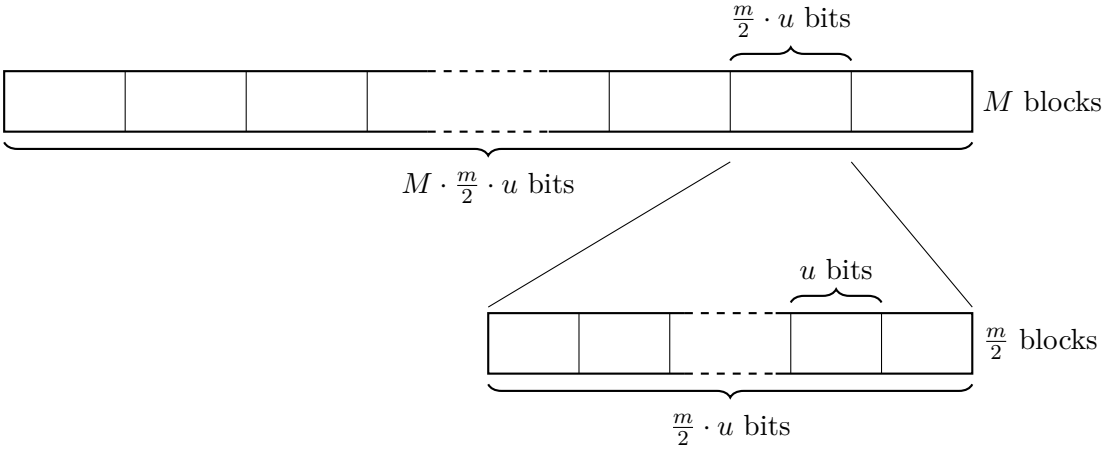

FIGURE 14: Encoding an input integer as a polynomial over $\mathcal{R}$

α as root of unity, since multiplications with powers of α can be performed as cyclic shifts and are thus cheap.

We now describe the process formally. By construction, $a(x) \in \mathcal{R}[x]$ is a polynomial with degree-bound $2M$ and $\rho \in \mathcal{R}$ is a principal $2M$-th root of unity. Bear in mind that $\rho^{2M/2m} = \rho^{\mu} = \alpha$. Since $\alpha^m \equiv -1$, α is a primitive $2m$-th root of unity in $\mathcal{R}$. We can compute the length-$2M$ DFT of $a(x)$ with ρ as root of unity in three steps:

i. Perform *inner DFTs*.[†]

Figure 15 shows the input vector a, which contains the coefficients of the polynomial $a(x)$. The arrow indicates the ordering of the elements for the DFT.

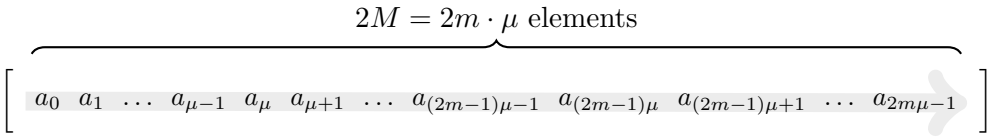

FIGURE 15: Input vector a

Rewrite the input vector a as $2m$ rows of μ columns and perform FFTs on the columns, see Figure 16. The boxes hold the values of vectors called e_ℓ, while the arrows indicate the ordering of their elements.

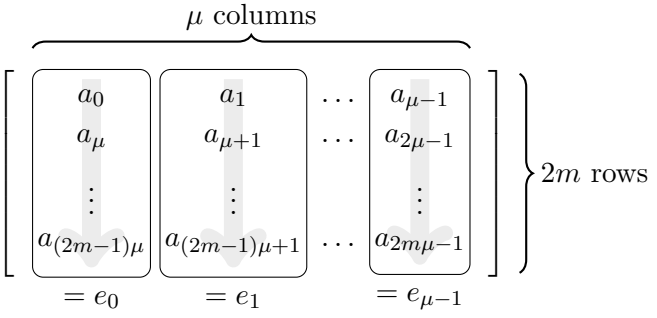

FIGURE 16: Input vector a written as μ column vectors of $2m$ elements

[†]Please note that the inner and outer *DFTs* have no relation to the inner or outer *coefficients*.

We now define polynomials $\bar{a}_v(x)$, which are residues of modular division. We will show that they can be calculated by performing DFTs on the e_ℓ.

Let $v \in [0 : 2m-1]$ and define polynomials $\bar{a}_v(x) \in \mathcal{R}[x]$ with degree-bound μ as

$$\bar{a}_v(x) \coloneqq a(x) \bmod (x^\mu - \alpha^v). \tag{3.4}$$

Denote $a_j \in \mathcal{R}$ the j-th coefficient of $a(x)$, let $\ell \in [0 : \mu - 1]$ and define $e_\ell(y) \in \mathcal{R}[y]$ as

$$e_\ell(y) \coloneqq \sum_{j=0}^{2m-1} a_{j\mu+\ell} \cdot y^j. \tag{3.5}$$

That is, the j-th coefficient of $e_\ell(y)$ is the $(j\mu+\ell)$-th coefficient of $a(x)$, and $e_\ell(y)$ is a polynomial over $\mathcal{R}$ with degree-bound $2m$.

To calculate $\bar{a}_v(x)$, write it out:

$$\begin{aligned}
\bar{a}_v(x) &= a(x) \bmod (x^\mu - \alpha^v) \\
&= (a_0 + a_1 x + \ldots + a_{\mu-1}x^{\mu-1} + \\
&\quad a_\mu x^\mu + a_{\mu+1}x^{\mu+1} + \ldots + a_{2\mu-1}x^{2\mu-1} + \\
&\quad a_{2\mu}x^{2\mu} + a_{2\mu+1}x^{2\mu+1} + \ldots + a_{3\mu-1}x^{3\mu-1} + \\
&\quad \ldots + a_{2M-1}x^{2M-1}) \bmod (x^\mu - \alpha^v).
\end{aligned}$$

Since $x^\mu \equiv \alpha^v \pmod{x^\mu - \alpha^v}$, replace x^μ with α^v and get

$$\begin{aligned}
\bar{a}_v(x) &= a_0 + a_1 x + \ldots + a_{\mu-1}x^{\mu-1} + \\
&\quad a_\mu\alpha^v + a_{\mu+1}\alpha^v x + \ldots + a_{2\mu-1}\alpha^v x^{\mu-1} + \\
&\quad a_{2\mu}\alpha^{2v} + a_{2\mu+1}\alpha^{2v} x + \ldots + a_{3\mu-1}\alpha^{2v} x^{\mu-1} + \\
&\quad \ldots + a_{2M-1}\alpha^{(2m-1)v} x^{\mu-1}.
\end{aligned}$$

Denote $\bar{a}_{v,\ell}$ the ℓ-th coefficient of $\bar{a}_v(x)$. Adding up coefficients of matching powers of x yields

$$\bar{a}_{v,\ell} = \sum_{j=0}^{2m-1} a_{j\mu+\ell} \cdot \alpha^{jv}.$$

Compare this to (3.5) to see that

$$\bar{a}_{v,\ell} = e_\ell(\alpha^v).$$

So to find the ℓ-th coefficient of each $\bar{a}_v(x)$ we can perform a length-$2m$ DFT of $e_\ell(y)$, using α as root of unity. Call these the *inner DFTs*. If we let ℓ run through its μ possible values, we get the coefficients of all $\bar{a}_v(x)$. Figure 17 shows the result of the inner DFTs.

$$\begin{aligned}
\bar{a}_0 &= \\
\bar{a}_1 &= \\
&\vdots \\
\bar{a}_{2m-1} &=
\end{aligned}
\left[\begin{array}{cccc}
\bar{a}_{0,0} & \bar{a}_{0,1} & \dots & \bar{a}_{0,\mu-1} \\
\bar{a}_{1,0} & \bar{a}_{1,1} & \dots & \bar{a}_{1,\mu-1} \\
\vdots & \vdots & & \vdots \\
\bar{a}_{2m-1,0} & \bar{a}_{2m-1,1} & \dots & \bar{a}_{2m-1,\mu-1}
\end{array}\right]$$

FIGURE 17: Result of inner DFTs as $2m$ row vectors of μ elements

Multiplications by powers of α can be performed as cyclic shifts. Since $\alpha^m \equiv -1$, coefficients of powers $\geq m$ wrap around with changed sign. This works much in the same way as the integer 2 in Schönhage and Strassen's multiplication algorithm in Section 2.9.

ii. Perform *bad multiplications*.

What De, Kurur, Saha and Saptharishi call *bad multiplications* is known as multiplications by twiddle factors in the Cooley-Tukey FFT.

Our goal is to compute the DFT of $a(x)$ with ρ as $2M$-th root of unity, that is, to compute $a(\rho^i)$, for $i \in [0 : 2M-1]$. Express i as $i = 2m \cdot f + v$ with $f \in [0 : \mu-1]$ and $v \in [0 : 2m-1]$. Then

$$a(\rho^i) = a(\rho^{2m \cdot f + v}) = \bar{a}_v(\rho^{2m \cdot f + v}), \tag{3.6}$$

because according to (3.4)

$$\bar{a}_v(\rho^{2m \cdot f + v}) = a(\rho^{2m \cdot f + v}) \bmod \underbrace{((\rho^{2m \cdot f + v})^\mu - \alpha^v)}_{=:\xi}$$

$$\begin{aligned}
\text{with} \quad \xi &= (\rho^{2m \cdot f + v})^\mu - \alpha^v \\
&= (\underbrace{\rho^{2M}}_{=1})^f \cdot (\underbrace{\rho^\mu}_{=\alpha})^v - \alpha^v \\
&= \alpha^v - \alpha^v \\
&= 0.^\dagger
\end{aligned}$$

We already computed the polynomials $\bar{a}_v(x)$ in Step i above. In order to efficiently compute $\bar{a}_v(\rho^{2m \cdot f + v})$, we define

$$\widetilde{a}_v(x) \coloneqq \bar{a}_v(x \cdot \rho^v), \tag{3.7}$$

so that if $\widetilde{a}_v(x)$ is evaluated at $x = \rho^{2m \cdot f}$ we get $\widetilde{a}_v(\rho^{2m \cdot f}) = \bar{a}_v(\rho^{2m \cdot f + v})$.

Computing $\widetilde{a}_v(x)$ can be done by computing its coefficients $\widetilde{a}_{v,\ell} = \bar{a}_{v,\ell} \cdot \rho^{v\ell}$, with $\ell \in [0 : \mu-1]$. Since coefficients are themselves polynomials, use Kronecker-Schönhage substitution as described in Section 3.2.6 to efficiently multiply them.

iii. Perform *outer DFTs*.

Now all that is left is to evaluate the $\widetilde{a}_v(x)$, $v \in [0 : 2m-1]$, at $x = \rho^{2m \cdot f}$, for $f \in [0 : \mu-1]$. In Step ii we arranged $\widetilde{a}_v(x)$ in such a way that this evaluation is nothing but a length-μ DFT of $\widetilde{a}_v(x)$ with ρ^{2m} as root of unity. Call these the *outer DFTs*. They are depicted in Figure 18.

[†] Since $a = b \pmod c$ means $a - b = kc$, for some $k \in \mathbb{Z}$, $a = b \pmod 0$ means equality.

$$\begin{bmatrix} \widetilde{a}_0 = & \widetilde{a}_{0,0} & \widetilde{a}_{0,1} & \dots & \widetilde{a}_{0,\mu-1} \\ \widetilde{a}_1 = & \widetilde{a}_{1,0} & \widetilde{a}_{1,1} & \dots & \widetilde{a}_{1,\mu-1} \\ & \vdots & \vdots & & \vdots \\ \widetilde{a}_{2m-1} = & \widetilde{a}_{2m-1,0} & \widetilde{a}_{2m-1,1} & \dots & \widetilde{a}_{2m-1,\mu-1} \end{bmatrix}$$

FIGURE 18: Outer DFTs on $2m$ row vectors of μ elements

If $M \geq m$ this is done by a recursive call to the FFT routine and according to (3.6) and (3.7) computes $\widetilde{a}_v(\rho^{2m \cdot f}) = \bar{a}_v(\rho^{2m \cdot f + v}) = a(\rho^{2m \cdot f + v}) = a(\rho^i)$.

If $M < m$, just computing an inner DFT with $\alpha^{2m/2M}$ as $(2m/2M)$-th root of unity is sufficient.

3.2.6 Componentwise Multiplication

Multiply coefficients $\widetilde{a}_i$ by $\widetilde{b}_i$ to compute $2M$ product coefficients $\widehat{c}_i := \widehat{a}_i \widehat{b}_i$. Since coefficients are from $\mathcal{R}$ and are thus themselves polynomials, we use Kronecker-Schönhage substitution (cf. [Sch82, sec. 2], [BZ11, sec. 1.3 & 1.9]) to multiply them and reduce polynomial multiplication to integer multiplication. Then we can use the DKSS algorithm recursively.

Definition (Kronecker-Schönhage substitution)**.** *Kronecker-Schönhage substitution reduces polynomial multiplication to integer multiplication. Since $\mathcal{R} = \mathcal{P}[\alpha]/(\alpha^m + 1)$ consists of polynomials with degree-bound m, whose coefficients are in $\mathcal{P} = \mathbb{Z}/p^z\mathbb{Z}$, each coefficient can be stored in $d := \lceil \log p^z \rceil$ bits. Coefficients are to be multiplied, so $2d$ bits per coefficient product must be allocated to prevent overflow. Furthermore, multiplication of two polynomials with degree-bound m leads to m summands for the middle coefficients, thus another $\log m$ bits per coefficient are required.*

This substitution converts elements of $\mathcal{R}$ into integers that are $m(2d + \log m)$ bits long. Then these integers are multiplied and from the result the product polynomial is recovered.

3.2.7 Backwards FFT

The backwards FFT works exactly like the forward FFT described in Step 5. We use in fact an inverse FFT and reordering and scaling of the resulting coefficients is handled in the next step.

3.2.8 Carry Propagation

In Step 4, we encoded an input number a into the polynomial $a(x)$ by putting $um/2$ bits into each outer coefficient and from there distributing u bits into each of the $m/2$ lower inner coefficients. When decoding the product polynomial $c(x)$ into the number c, we must use the same weight as for encoding, so we evaluate the inner coefficients at $\alpha = 2^u$ and the outer coefficients at $x = 2^{um/2}$. Of course, on a binary computer this evaluation can be done by bit-shifting and addition.

We must take the ordering of the resulting coefficients into account. In Section 2.7 we defined a backwards transform to get results that are properly ordered. However, for simplicity of

implementation, we use again a forward transform and access its resulting coefficients in different order.

Furthermore, all result coefficients are scaled by a factor of $2M$, so we have to divide them by $2M$ prior to addition.

3.3 Run-time Analysis

3.3.1 Analysis of each Step

Our goal is to find an upper bound to the bit complexity $T(N)$ needed to multiply two non-negative N-bit integers using the implementation of DKSS multiplication to get their $2N$-bit product. We estimate the run-time of each step individually.

1. Choosing M and m does only depend on the length of the input and can be done in constant time.

2. Computing u takes constant time as does finding p, since we precomputed all values for p for the supported hardware. Thus, this step has cost $O(1)$ as well.

3. In this step we compute a $2M$-th root of unity $\rho \in \mathcal{R}$ from a known generator ζ of $\mathbb{F}_p^*$. $T_\mathcal{P}$ denotes the time to multiply two arbitrary numbers in $\mathcal{P}$. First, we use Hensel lifting to calculate ζ_z in $z-1$ lifting steps. In each step we have to calculate

$$\zeta_{s+1} \coloneqq \zeta_s - (\zeta_s^{p-1} - 1) \cdot ((p-1)\zeta_s^{p-2})^{-1}.$$

   This can be done with 1 exponentiation, 3 multiplications, 4 subtractions and 1 modular inverse.

   To exponentiate, we use *binary exponentiation* [Knu97b, ch. 4.6.3], which requires $O(\log p)$ multiplications in $\mathcal{P}$, and to find the modular inverse we use the *extended Euclidean algorithm* [Knu97b, ch. 4.5.2] with $O(\log p^z)$ steps, where each step costs $O(\log p^z)$. After lifting, we calculate $\omega = \zeta_z^h$, where $h < p/2M$.

   Together, the cost T_ω to calculate ω is

$$\begin{aligned} T_\omega &= (z-1)\big(O(\log p)T_\mathcal{P} + 3T_\mathcal{P} + 4O(\log p^z) + O(\log p^z)O(\log p^z)\big) + \\ &\quad O(\log p)T_\mathcal{P} \\ &= O(\log p \cdot T_\mathcal{P} + \log^2 p^z). \end{aligned}$$

   After that, we perform Lagrange interpolation: according to (3.2) it consists of m additions of polynomials in $\mathcal{R}$, each of which is computed by $m-1$ multiplications of degree-1 polynomials with polynomials in $\mathcal{R}$ plus $m-1$ modular inverses in $\mathcal{P}$.

   Thus, the run-time for Lagrange interpolation is

$$\begin{aligned} T_{Lagrange} &= m(mO(\log p^z) + (m-1)(2mT_\mathcal{P} + O(\log p^z \cdot \log p^z))) \\ &= O(m^2(\log p^z + mT_\mathcal{P} + \log^2 p^z)) \\ &= O(m^2(mT_\mathcal{P} + \log^2 p^z)). \end{aligned}$$

Ordinary multiplication can multiply n-bit integers in run-time $O(n^2)$, hence $T_{\mathcal{P}}$ can be bounded by $O(\log^2 p^z)$. Using (3.1) we estimate $p^z = O(N^{Lz}/\log^{2Lz} N)$ and recall that $m = O(\log N)$. We get as total time to compute ρ:

$$\begin{aligned}
T_\rho &= T_\omega + T_{Lagrange} \\
&= O(\log p \cdot T_{\mathcal{P}} + \log^2 p^z)) + O(m^2(mT_{\mathcal{P}} + \log^2 p^z)) \\
&= O(\log p \cdot O(\log^2 p^z) + \log^2 p^z + m^2(mO(\log^2 p^z) + \log^2 p^z)) \\
&= O(\log p \cdot \log^2 p^z + m^2(m \log^2 p^z + \log^2 p^z)) \\
&= O((\log p + m^3) \log^2 p^z) \\
&= O((\log(N^L/\log^{2L} N) + \log^3 N) \log^2(N^{Lz}/\log^{2Lz} N)) \\
&= O(\log^3 N \cdot \log^2 N) \\
&= O(\log^5 N).
\end{aligned}$$

4. Encoding input numbers as polynomials can be done in time proportional to the length of the numbers, that is, in time $O(N)$.

5. As we will see, the FFT is one of the two most time-consuming steps; the other one being the multiplication of sample values. Let us first evaluate the run-time of a length-$2M$ FFT over $\mathcal{R}$, denoted by $T_D(2M)$. We analyze the run-time of each step of the FFT individually. $T_{\mathcal{R}}$ denotes the time needed to multiply two arbitrary elements of $\mathcal{R}$ and will be specified later.

   i. The first step performs $\mu = 2M/2m$ inner FFTs over $\mathcal{R}$ of length $2m$. To calculate one DFT we need to perform $2m \log(2m)$ additions and $m \log(2m)$ multiplications by powers of α, cf. (2.4). A single addition costs $O(m \log p^z)$, since an element of $\mathcal{R}$ is a polynomial over $\mathcal{P} = \mathbb{Z}/p^z\mathbb{Z}$ with degree-bound m. Since multiplication by a power of α can be done with a cyclic shift, its run-time is of the same order as that of an addition. So the run-time to compute one inner DFT is

   $$3m \log(2m) \cdot O(m \log p^z) = O(m^2 \log m \cdot \log p^z),$$

   and the run-time to compute all $2M/2m$ inner DFTs in this step is

   $$2M/2m \cdot O(m^2 \log m \cdot \log p^z) = O(Mm \log m \cdot \log p^z).$$

   ii. Here, we prepare the $2m$ polynomials $\bar{a}_v(x)$ for the outer DFTs. For each $v \in [0 : 2m - 1]$, the polynomial $\bar{a}_v(x)$ has $\mu = 2M/2m$ coefficients, which makes a total of $2M$ multiplications in $\mathcal{R}$ by powers of ρ to compute all $\widetilde{a}_v(x)$. The same number of multiplications is needed to compute the powers of ρ. So this step has a total run-time of $4M \cdot T_{\mathcal{R}}$.

   iii. This last step computes $2m$ outer DFTs. The FFT routine is invoked recursively to do this. The total run-time for this step is the time for $2m$ DFTs of length $2M/2m$ and hence

   $$2m \cdot T_D(2M/2m).$$

   The recursion stops when the FFT length is $\leq 2m$, that is, after $\log_{2m}(2M)$ levels.

The total run-time $T_D(2M)$ of the FFT is the sum of the run-times of all three steps, that is,

$$\begin{aligned} T_D(2M) &= O(Mm \log m \cdot \log p^z) + 4M \cdot T_{\mathcal{R}} + 2m \cdot T_D(2M/2m) \\ &= \log_{2m}(2M) \cdot \big(O(Mm \log m \cdot \log p^z) + 4M \cdot T_{\mathcal{R}}\big). \end{aligned} \tag{3.8}$$

6. Each of the $2M$ coefficient pairs $\widehat{a_i}$, $\widehat{b_i}$ can be multiplied in time $T_{\mathcal{R}}$. Thus, the run-time for this step is $2M \cdot T_{\mathcal{R}}$.

7. The backwards FFT has the same cost as the forward FFT, see (3.8).

8. Decoding the polynomials back into integers and performing carry propagation can be done with $2Mm$ additions of length $\log p^z$, hence with cost

$$\begin{aligned} T_{decode} &= O(2Mm \log p^z) \\ &= O\Big(\frac{N}{\log^2 N} \log N \cdot \log \frac{N^{Lz}}{\log^{2Lz} N}\Big) \\ &= O\Big(\frac{N}{\log N} \log N^{Lz}\Big) \\ &= O(N). \end{aligned}$$

3.3.2 Putting Everything Together

To conclude our evaluation of the run-time, we need to upper bound the value of $T_{\mathcal{R}}$, the time needed to multiply two arbitrary elements of $\mathcal{R}$. For that purpose, we use Kronecker-Schönhage substitution as described in Section 3.2.6.

Theorem (Kronecker-Schönhage substitution)**.** *Multiplication in $\mathcal{R}$ can be reduced to integer multiplication of length $O(\log^2 N)$ bits.*

This substitution converts elements of $\mathcal{R}$ into integers of $m(2d + \log m)$ bits, with $d = \lceil \log p^z \rceil$, multiplies the integers and from the integer result recovers the product polynomial. To see how large these integers get in terms of N, we use (3.1) and obtain

$$\begin{aligned} m(2d + \log m) &= O(\log N \cdot (2\lceil \log p^z \rceil + \log\log N)) \\ &= O(\log N \cdot (\log p^z + \log\log N)) \\ &= O(\log N \cdot (\log(N^{Lz}/\log^{2Lz} N) + \log\log N)) \\ &= O(\log N \cdot Lz \log N) \\ &= O(\log^2 N). \end{aligned} \tag{3.9}$$

$T(N)$ denotes the time to multiply two N-bit integers, so $T_{\mathcal{R}} = T(O(\log^2 N))$. □

Adding up the run-time estimates of all steps we get the total run-time $T(N)$ for DKSS multiplication:

$$\begin{aligned}
T(N) &= O(1) + O(1) + O(\log^5 N) + O(N) + 2T_D(2M) + 2MT_{\mathcal{R}} + T_D(2M) + O(N) \\
&= 3T_D(2M) + 2MT_{\mathcal{R}} + O(N) \\
&= 3\big(\log_{2m}(2M)(O(Mm\log m \cdot \log p^z) + 4MT_{\mathcal{R}})\big) + 2MT_{\mathcal{R}} + O(N) \\
&= O\big(\log_{2m}(2M)(Mm\log m \cdot \log p^z + MT_{\mathcal{R}}) + MT_{\mathcal{R}} + N\big) \\
&= O(M\log_{2m}(2M)(m\log m \cdot \log p^z + T(O(\log^2 N))) + MT(O(\log^2 N)) + N).
\end{aligned}$$

In terms of N that is

$$\begin{aligned}
T(N) &= O\Big(\frac{N}{\log^2 N}\frac{\log(2N/\log^2 N)}{\log(2\log N)}\big(\log N \cdot \log\log N \cdot \log\big(\frac{N^{Lz}}{\log^{2Lz} N}\big) + \\
&\quad T(O(\log^2 N))\big) + \frac{N}{\log^2 N}T(O(\log^2 N)) + N\Big) \\
&= O\Big(\frac{N}{\log^2 N}\frac{\log N}{\log\log N}\big(\log N \cdot \log\log N \cdot \log N + T(O(\log^2 N))\big) + \\
&\quad \frac{N}{\log^2 N}T(O(\log^2 N)) + N\Big) \\
&= O\Big(N\log N + \frac{N}{\log N \cdot \log\log N}T(O(\log^2 N)) + \frac{N}{\log^2 N}T(O(\log^2 N)) + N\Big) \\
&= O\Big(N\log N + \frac{N}{\log N \cdot \log\log N} \cdot T(O(\log^2 N))\Big). \qquad (3.10)
\end{aligned}$$

3.3.3 Resolving the Recursion

To solve the recursion, we will need the following estimation. Observe that for any real $x \geq 4$ it holds that

$$\frac{\log(\lambda\log^2 x)}{\log\log x} = \frac{\log(\lambda(\log x)^2)}{\log\log x} = \frac{\log\lambda + 2\log\log x}{\log\log x} \leq \log\lambda + 2. \qquad (3.11)$$

The following notation is introduced to abbreviate the upcoming nested logarithms: define $f_0(N) := N$ and $f_i(N) := \lambda\log^2 f_{i-1}(N)$, for $i \in \mathbb{N}$ and some λ. Furthermore, let $\tau \geq 4$ be the smallest length where the algorithm is used, otherwise a simpler algorithm is used. Now we express the run-time from (3.10) with explicit constants, assuming that $\lambda\log^2 N = f_1(N) \geq \tau$ and unroll the recursion once:

$$\begin{aligned}
T(N) &\leq \mu(N\log N + \frac{N}{\log N \cdot \log\log N}T(\lambda\log^2 N)) \\
&= \mu N\log N\Big(1 + \frac{T(\lambda\log^2 N)}{\log^2 N \cdot \log\log N}\Big) \\
&\leq \mu N\log N\Big(1 + \frac{\mu(\lambda\log^2 N)\log(\lambda\log^2 N)}{\log^2 N \cdot \log\log N}(1 + \frac{T(\lambda\log^2(\lambda\log^2 N))}{\log^2(\lambda\log^2 N) \cdot \log\log(\lambda\log^2 N)})\Big) \\
&= \mu N\log N\Big(1 + \frac{\mu\lambda\log(\lambda\log^2 N)}{\log\log N}(1 + \frac{T(\lambda\log^2 f_1(N))}{\log^2 f_1(N) \cdot \log\log f_1(N)})\Big).
\end{aligned}$$

Using (3.11) leads to

$$\begin{aligned} T(N) &\leq \mu N \log N \Big(1 + \underbrace{\mu\lambda(\log\lambda + 2)}_{=:\eta}(1 + \frac{T(\lambda \log^2 f_1(N))}{\log^2 f_1(N) \cdot \log\log f_1(N)})\Big) \\ &= \mu N \log N \Big(1 + \eta + \eta \cdot \frac{T(\lambda \log^2 f_1(N))}{\log^2 f_1(N) \cdot \log\log f_1(N)}\Big). \end{aligned}$$

Assuming $\lambda \log^2 f_1(N) = f_2(N) \geq \tau$ we unroll once more and get

$$\begin{aligned} T(N) \leq \mu N \log N \Big(1 + \eta + \\ \eta \cdot \frac{\mu\lambda \log^2 f_1(N) \cdot \log(\lambda \log^2 f_1(N))}{\log^2 f_1(N) \cdot \log\log f_1(N)}(1 + \frac{T(\lambda \log^2 f_2(N))}{\log^2 f_2(N) \cdot \log\log f_2(N)})\Big). \end{aligned}$$

Again canceling out and using (3.11) gives

$$\begin{aligned} T(N) &\leq \mu N \log N \Big(1 + \eta + \eta \underbrace{\mu\lambda(\log\lambda + 2)}_{=\eta}(1 + \frac{T(\lambda \log^2 f_2(N))}{\log^2 f_2(N) \log\log f_2(N)})\Big) \\ &= \mu N \log N \Big(1 + \eta + \eta^2 + \eta^2 \cdot \frac{T(\lambda \log^2 f_2(N))}{\log^2 f_2(N) \log\log f_2(N)}\Big) \\ &= \mu N \log N \Big(\sum_{i=0}^{2} \eta^i + \eta^2 \cdot \frac{T(\lambda \log^2 f_2(N))}{\log^2 f_2(N) \log\log f_2(N)}\Big). \end{aligned}$$

Obviously, after unrolling $j \in \mathbb{N}_0$ levels of recursion and assuming $f_j(N) \geq \tau$ we get

$$T(N) \leq \mu N \log N \Big(\sum_{i=0}^{j} \eta^i + \eta^j \cdot \frac{T(\lambda \log^2 f_j(N))}{\log^2 f_j(N) \log\log f_j(N)}\Big). \tag{3.12}$$

The remaining question is now: how many levels of recursion are there for a given N? To find out, we look for a lower bound for N after j levels of recursion.

Equation (3.12) applies if $f_j(N) \geq \tau$. If $j \geq 1$ we can reduce $f_j(N)$ once and get

$$\begin{aligned} f_j(N) &\geq \tau \\ \lambda \log^2 f_{j-1}(N) &\geq \tau \\ \log^2 f_{j-1}(N) &\geq \tau/\lambda \\ \log f_{j-1}(N) &\geq \sqrt{\tau/\lambda} \\ f_{j-1}(N) &\geq 2^{\sqrt{\tau/\lambda}}. \end{aligned} \tag{3.13}$$

A second reduction works quite like the first, assuming $j \geq 2$:

$$\begin{aligned} \lambda \log^2 f_{j-2}(N) &\geq 2^{\sqrt{\tau/\lambda}} \\ \log f_{j-2}(N) &\geq \sqrt{2^{\sqrt{\tau/\lambda}}/\lambda} \\ f_{j-2}(N) &\geq 2^{\sqrt{2^{\sqrt{\tau/\lambda}}/\lambda}}. \end{aligned}$$

Transforming the exponent we get

$$\sqrt{2^{\sqrt{\tau/\lambda}}/\lambda} = \sqrt{(2^{\sqrt{1/\lambda}})^{\sqrt{\tau}}/\lambda} = \sqrt{(2^{\sqrt{1/\lambda}})^{\sqrt{\tau}}}/\sqrt{\lambda} = \underbrace{(2^{\sqrt{1/\lambda}})^{\frac{1}{2}\cdot\sqrt{\tau}}}_{=:\beta}/\sqrt{\lambda} = \beta^{\sqrt{\tau}}/\sqrt{\lambda}.$$

Now use that and reduce again, assuming $j \geq 3$:

$$\begin{aligned} f_{j-2}(N) &\geq 2^{\beta^{\sqrt{\tau}}/\sqrt{\lambda}} \\ \lambda \log^2 f_{j-3}(N) &\geq 2^{\beta^{\sqrt{\tau}}/\sqrt{\lambda}} \\ f_{j-3}(N) &\geq 2^{\sqrt{2^{\beta^{\sqrt{\tau}}/\sqrt{\lambda}}/\lambda}}. \end{aligned}$$

Transforming the exponent again gives

$$\sqrt{2^{\beta^{\sqrt{\tau}}/\sqrt{\lambda}}/\lambda} = \sqrt{(2^{\sqrt{1/\lambda}})^{\beta^{\sqrt{\tau}}}/\lambda} = \sqrt{(2^{\sqrt{1/\lambda}})^{\beta^{\sqrt{\tau}}}}/\sqrt{\lambda} = \underbrace{(2^{\sqrt{1/\lambda}})^{\frac{1}{2}\cdot\beta^{\sqrt{\tau}}}}_{=\beta}/\sqrt{\lambda} = \beta^{\beta^{\sqrt{\tau}}}/\sqrt{\lambda},$$

which yields

$$f_{j-3}(N) \geq 2^{\beta^{\beta^{\sqrt{\tau}}}/\sqrt{\lambda}}. \tag{3.14}$$

So we see that with each unroll step of $f_j(N)$ we get another exponentiation by β in the exponent.

Definition (Iterated Exponentials)**.** *Let a, $x \in \mathbb{R}$, $n \in \mathbb{N}_0$ and denote $\exp_a(x) = a^x$, then*

$$\exp_a^n(x) := \begin{cases} x & \text{if } n = 0 \\ \exp_a(\exp_a^{n-1}(x)) & \text{if } n > 0 \end{cases}$$

is called iterated exponentials *or* power tower*. For example, $\exp_a^3(x) = a^{a^{a^x}}$. This notation is inspired by Euler's $\exp(x)$ function and* functional iteration *in [CLRS09, p. 58].*

With the help of iterated exponentials we can write (3.14) as

$$f_{j-3}(N) \geq 2^{\exp_\beta^2(\sqrt{\tau})/\sqrt{\lambda}},$$

and if we reduce $f_j(N)$ fully we get

$$N = f_0(N) \geq 2^{\exp_\beta^{j-1}(\sqrt{\tau})/\sqrt{\lambda}}. \tag{3.15}$$

We are close to the goal, which we can attain with help of the following

Definition (Iterated Logarithm [CLRS09, p. 58])**.** *Let a, $x \in \mathbb{R}_{>0}$, then the* iterated logarithm *is defined as*

$$\log_a^*(x) := \begin{cases} 0 & \text{if } x \leq 1 \\ \log_a^*(\log_a x) + 1 & \text{if } x > 1 \end{cases} \tag{3.16}$$

and is the inverse of $\exp_a^n(1)$, that is, $\log_a^*(\exp_a^n(1)) = n$. The iterated logarithm is the number of $\log_a$-operations needed to bring its argument to a value ≤ 1. As usual, $\log^* x := \log_2^* x$.

Now we use the iterated logarithm on (3.15) and get

$$\begin{aligned} N &\geq 2^{\exp_\beta^{j-1}(\sqrt{\tau})/\sqrt{\lambda}} \\ \log N &\geq \exp_\beta^{j-1}(\sqrt{\tau})/\sqrt{\lambda} \\ \sqrt{\lambda}\log N &\geq \exp_\beta^{j-1}(\sqrt{\tau}) \\ \log_\beta^*(\sqrt{\lambda}\log N) &\geq j - 1 + \log_\beta^* \sqrt{\tau} \\ \log_\beta^*(\sqrt{\lambda}\log N) + 1 - \log_\beta^* \sqrt{\tau} &\geq j. \end{aligned} \tag{3.17}$$

We can replace $\log_\beta^* x$ by $O(\log_2^* x)$. To see why, observe that if β could be expressed as some power tower of 2, say, $\beta = 2^{2^2}$, that is, $\log^* \beta = 3$, then a power tower of β is less than one of 2 with thrice the length, because $\beta^\beta = (2^{2^2})^\beta < 2^{2^{(2^\beta)}}$. Hence, $\log_\beta^* x \leq \log^* x \cdot \log^* \beta = O(\log^* x)$, since β is constant.

Since only N is a variable, this finally leads to the estimate

$$\begin{aligned} j &\leq \log_\beta^*(\sqrt{\lambda}\log N) + 1 - \log_\beta^* \sqrt{\tau} \\ &= O(\log_\beta^*(\sqrt{\lambda}\log N)) \\ &= O(\log_\beta^* N) \\ &= O(\log^* N). \end{aligned} \tag{3.18}$$

Now, we can pick up (3.12) again. We assume $\eta \neq 1$, $f_j(N) \geq \tau$, but $f_{j+1}(N) < \tau$ and hence in analogy to (3.13), $f_j(N) < 2^{\sqrt{\tau/\lambda}}$. Then we get

$$\begin{aligned} T(N) &\leq \mu N \log N \Big(\sum_{i=0}^{j} \eta^i + \eta^j \cdot \frac{T(\lambda \log^2 f_j(N))}{\log^2 f_j(N) \log\log f_j(N)} \Big) \\ &\leq \mu N \log N \Big(\frac{\eta^{j+1} - 1}{\eta - 1} + \eta^j \cdot \frac{T(f_{j+1}(N))}{\log^2 f_j(N) \log\log f_j(N)} \Big). \end{aligned}$$

Capturing the constants into Big-O's yields

$$\begin{aligned} T(N) &= \mu N \log N (O(\eta^{j+1}) + O(\eta^j)) \\ T(N) &= O(N \log N \cdot \eta^{j+1}) \\ &= N \log N \cdot \eta^{O(\log^* N)}. \end{aligned}$$

Expressing η and the constant from $O(\ldots)$ as 2^κ, for some constant κ, we write

$$\begin{aligned} T(N) &= N \log N \cdot (2^\kappa)^{O(\log^* N)} \\ &= N \cdot \log N \cdot 2^{O(\log^* N)}. \end{aligned} \tag{3.19}$$

3.4 Differences to DKSS Paper

The intention of this thesis is to assess the speed of an implementation of DKSS multiplication on a modern computer. Its architecture imposes certain limits on its software. For example, the amount of memory that can be addressed is limited by the size of the processor's index registers. A more compelling limit is that the universe contains only a finite amount of matter

and energy, as far as we know. A computer will need at least one electron per bit and thus, even if we could harness all (dark) matter and energy for memory bits, any storable number could surely not exceed 2^{300} bits in length.

Another limit creeps in with the speed of the machine: there is no practical use to provide a solution that will run several thousand years or more to complete. An estimation of the run-time to multiply numbers with 2^{65} bits leads to a minimum of 7000 years on the test machine.

This led me to assume a maximum length of input numbers. Since the implementation runs on a 64-bit CPU, the number's length is de facto limited to $8 \cdot 2^{64}/4 = 2^{65}$ bits. And since the length is limited, we can precompute some constants needed in the algorithm, namely the prime p and a generator ζ of $\mathbb{F}_p^*$. I did this for values of p with 2 to 1704 bits in length.

De, Kurur, Saha and Saptharishi went to great lengths to show that suitable primes p can be found at run-time and to make their construction work, they use p^z as modulus, $z > 1$, as we have seen in Sections 3.2.2 and 3.2.3.

Furthermore, they encode input numbers as k-variate polynomials, where the degree in each variable is $< 2M$. That is, outer polynomials are in $\mathcal{R}[X_1, \ldots, X_k]$. When it comes to the FFT, they fix one variable, say X_k, and treat the outer polynomials as univariate polynomials over $\mathcal{S} := \mathcal{R}[X_1, \ldots, X_{k-1}]$. Note that ρ is a principal $2M$-th root of unity in $\mathcal{S}$ as well. Then they perform FFT multiplication of a univariate polynomial over $\mathcal{S}$. The componentwise multiplication uses FFT multiplication recursively, because now two $(k-1)$-variate polynomials have to be multiplied.

Since the only need for k-variate polynomials was to show that p can be found at run-time, I was able to use $k = 1$ and use univariate polynomials in the implementation. Furthermore, it was easy to precompute p to greater sizes, so there was no need for $z > 1$ and thus I dropped Hensel lifting to find ζ_z as well.

I changed some variable names from [DKSS13] to avoid confusion with other variables of the same name or to improve clarity. If the reader is familiar with the original paper, here is a small overview of changed names:

Description	DKSS paper	This thesis
Exponent of prime p in modulus	c	z
Number of variables for outer polynomials	k	(dropped, $k = 1$)
Factor in progression for finding prime p	i	h
Residue polynomials in DFT	a_j	$\bar{a}_v$
Index variable in DFT	k	f
Radix of FFT	$2M/2m$	μ

Chapter 4

Implementation of DKSS Multiplication

In this chapter my implementation of DKSS multiplication is presented. Parameter selection is discussed and exemplary source code is shown, together with a description of tests performed to assert the software's correctness. Then, measured execution time, memory requirements and source code size is examined. I discuss the results of profiling and lastly, extrapolate run-time for increasing input lengths.

4.1 Parameter Selection

The description of parameter selection in Section 3.2 leaves some freedom on how exactly to calculate M, m, u and p. Recall that we are performing FFTs of polynomials with degree-bound $2M$ in $\mathcal{R}[x]$, where $\mathcal{R} = \mathcal{P}[\alpha]/(\alpha^m + 1)$ and $\mathcal{P} = \mathbb{Z}/p^z\mathbb{Z}$. We call coefficients in $\mathcal{R}[x]$ *outer coefficients* and coefficients in $\mathcal{P}[\alpha]$ *inner coefficients*. Both input numbers have N bits, parameter u is the number of bits of the input number that go into each inner coefficient and z is constant.

I aimed at a monotonically increasing graph of execution time, that is, growing input lengths lead to growing execution times. Parameter selection that leads to a rough graph suggests that better parameters could be selected.

This led me to choose the prime p first. Section 3.2.2 mentions lower bounds for p^z. Recall that $M \approx N/\log^2 N$ and $m \approx \log N$. I use

$$p^z \geq \frac{1}{2} M m 2^{2u} \approx \frac{1}{2} N^5 / \log N. \tag{4.1}$$

Furthermore, I decided to round up the number of bits of p to the next multiple of the word size. Since both allocated memory and cost of division (for modular reductions) depend on the number of words, it seemed prudent to make the most out of it. Benchmarks show that this was a good choice, see Figure 19 for a graph of timings.

DKSS multiplication uses $\mathcal{P} = \mathbb{Z}/p^z\mathbb{Z}$ with $z > 1$ to lower run-time in the asymptotic case by lowering the upper bound for finding the prime p. But that doesn't apply here, since the machine this implementation runs on enforces upper limits of the length of numbers. So despite

the description of the process of Hensel lifting in Section 3.2.3, I did not implement it, because precomputation of larger prime numbers was the easier choice (see Linnik's Theorem on page 36). Furthermore, the special build of Proth prime numbers could be exploited in the future to speed up modular reductions.

Having chosen p, I then select the largest u that is able to hold the whole $2N$ bits of the result. It follows from (4.1) that $\log(p^z) \geq \log(Mm) + 2u - 1$. Since $\log(p^z)$ is chosen first, I try to maximize u. The larger u is, the less coefficients are needed. After finding an u that fits, I try to minimize the product Mm, because the smaller Mm is, the smaller the FFT length and the memory requirements are.

Lastly, I set M and m and try to maintain the quotient $M/m \approx N/\log^3 N$ that follows from the description in Section 3.2.1. On the other hand, factors can be moved around between M and m, since in selection of u and p only the product Mm is needed. I did some short tests on selecting $M/m \approx k \cdot N/\log^3 N$ for some k, but it seemed that $k = 1$ was overall a good choice.

4.2 A Look at the Code

If the parameters are given (namely M, m, u, p^z and ρ), the main idea of DKSS multiplication lies in the structure of the ring $\mathcal{R}$ and the way the DFT is computed: inner DFTs, bad multiplications and outer DFTs.

To give an impression of the implementation, following is the FFT main routine. Language constructs (like templates and `typedef`s), debug code and assertions were stripped to improve readability. As mentioned in Section 2.2, `tape_alloc` is a stack-like memory allocator. It takes the number of `word`s requested as argument.

```
void dkss_fft(
   word* a,              // input vector
   unsigned M,
   unsigned m,
   unsigned oclen,       // outer coeff length = m * iclen
   unsigned iclen,       // inner coeff length >= bit_length(p^z) / bits(word)
   word* pz,             // modulus p^z
   word* rho_pow,        // powers [0 : m-1] of \rho
   unsigned base_pow)    // quotient of order of top level \rho and now
{
   if (M <= m) {         // use inner DFT right away
      tape_alloc tmp(oclen);              // allocate some memory from the "tape"
      word* t = tmp.p;                    // length oclen
      unsigned log_M = int_log2(M);
      fft_shuffle(a, log_M + 1, oclen);   // pre-shuffle the values
      dkss_inner_fft_eval(a, M, m, oclen, iclen, pz, t);
      return;
   }

   unsigned mu = M / m;                        // \mu = 2M/2m
   unsigned log2m = int_log2(m) + 1;
   tape_alloc tmp(2*M * oclen + 2*m * oclen + oclen);
   word* abar = tmp.p;                         // length 2*M*oclen, \bar{a}
   word* el = abar + 2*M * oclen;              // length 2*m*oclen, e_\ell
   word* t = el + 2*m*oclen;                   // length oclen, temp storage

   // perform inner DFTs
   // i guess it's better to copy elements instead of using pointers and work
   // in-place, because this way cache thrashing can only occur once when
   // copying and not on every access.
   for (unsigned l=0; l<mu; ++l) {             // cycle through all values for l
      // assemble e_l(y):
```

```
      // the j-th coeff of e_l(y) is the (j*mu+l)-th coeff of a(x)
      // for the FFT evaluation, we assemble e_l(y) already in shuffled order
      word* a_jxl = a + l*oclen;              // points to a_l
      for (unsigned j=0; j<2*m; ++j) {
         word* el_j = el + bit_rev(j, log2m) * oclen;
         copy(el_j, a_jxl, oclen);
         a_jxl += mu * oclen;                 // point to next a_{j*\mu+l}
      }

      // perform inner DFT on e_l(y) with alpha as 2m-th root of unity
      dkss_inner_fft_eval(el, m, m, oclen, iclen, pz, t);

      // l-th coeffs of all a_v(x) is e_l(alpha^v), i.e. v-th coeff of DFT(e_l)
      // this copies transformed elements back into place
      word* el_v = el;
      word* abar_vl = abar + l*oclen;
      for (unsigned v=0; v<2*m; ++v) {
         copy(abar_vl, el_v, oclen);
         el_v += oclen;
         abar_vl += mu * oclen;
      }
   }

   // perform bad muls and outer DFTs
   word* abar_v = abar;
   word* rho_vl = t;                          // just for the name
   const index top_mu = mu * base_pow;        // top level mu
   unsigned psh = int_log2(top_mu);           // shift count
   // cycle through all a_v to perform bad muls and outer DFTs
   for (unsigned v=0; v<2*m; ++v) {
      // skip first loop iteration: v == 0, i.e. abar_{v,l} *= rho^0 = 1
      word* abar_vl = abar_v;
      unsigned vlbase = 0;
      for (unsigned l=1; l<mu; ++l) {         // cycle thru all values for l
         vlbase += v * base_pow;
         abar_vl += oclen;
         unsigned pi = vlbase & ((1 << psh) - 1);     // vlbase % top_mu
         unsigned pe = vlbase >> psh;                 // vlbase / top_mu
         // select right rho_pow and do cyclic shift
         modpoly_mul_xpow_mod_mp1(rho_vl, rho_pow + pi*oclen, pe, m, iclen, pz);
         // abar_{v,l} *= rho^{vl}
         modpoly_mul_mod_mp1(abar_vl, abar_vl, rho_vl, m, iclen, pz);
      }

      // now abar_v contains \tilde{a}_v.  ready to do outer DFT: recursive call
      dkss_fft(abar_v, mu/2, m, oclen, iclen, pz, rho_pow, base_pow * 2*m);

      // copy back to 'a' array
      word* a_fxv = a + v * oclen;
      word* abar_vf = abar_v;
      for (unsigned f=0; f<mu; ++f) {
         copy(a_fxv, abar_vf, oclen);
         abar_vf += oclen;
         a_fxv += 2*m * oclen;
      }

      abar_v += mu * oclen;
   }
}
```

The listing shows one of the few optimizations I was able to implement: in the run-time analysis in Section 3.3, Step 5.ii we counted $2M$ multiplications by powers of ρ and another $2M$ multiplications to compute those powers. I was able to reduce the number of multiplications for the latter from $2M$ to $\mu = 2M/2m$.

I used the fact that $\rho^{2M/2m} = \rho^{\mu} = \alpha$: if $i \in [0 : 2M-1]$, set $r := \lfloor i/\mu \rfloor$ and $s := i \bmod \mu$ and thus $i = r\mu + s$.

Therefore it holds that $\rho^i = \rho^{r\mu+s} = \rho^{\mu \cdot r}\rho^s = \alpha^r\rho^s$. We can obtain ρ^i with an additional cyclic shift by precomputing all ρ^s, $s \in [0 : \mu - 1]$. In benchmarks, this almost halved the run-time.

In the above listing function `dkss_inner_fft_eval()` is called. This function doesn't differ much from the `QMUL` FFT evaluate function `qmul_evaluate()` on page 20, except that this time functions instead of operators are used to add and subtract elements, and multiplications by powers of the root of unity are done by cyclic shifts. Following is the listing of `dkss_inner_fft_eval()`:

```
void dkss_inner_fft_eval(
   word* e,                 // input vector
   unsigned n_half,         // half of FFT length
   unsigned m,
   unsigned oclen,          // outer coeff length = m * iclen
   unsigned iclen,          // inner coeff length >= bit_length(pz) / bits(word)
   word* pz,                // p^z
   word* t)                 // temp storage
{
   if (n_half == 1) {
      // lowest layer: butterfly of two outer coeffs,
      // i.e. add and sub of two inner polynomials
      word* e2 = e + oclen;                  // second inner polynomial
      copy(t, e2, oclen);
      modpoly_sub(e2, e, t, m, iclen, pz);   // e2 = e - t
      modpoly_add(e, e, t, m, iclen, pz);    // e = e + t
      return;
   }

   dkss_inner_fft_eval(e, n_half/2, m, oclen, iclen, pz, t);
   dkss_inner_fft_eval(e + n_half*oclen, n_half/2, m, oclen, iclen, pz, t);

   unsigned inc = m / n_half;                // increment for each loop
   word* e1 = e;                             // first inner polynomial
   word* e2 = e + n_half*oclen;              // second inner polynomial
   unsigned pow = 0;
   for (unsigned i=0; i<n_half; ++i) {
      // w = omega_n^i, t = w*e2
      modpoly_mul_xpow_mod_mp1(t, e2, pow, m, iclen, pz);  // cyclic shift by pow
      modpoly_sub(e2, e1, t, m, iclen, pz);  // e2 = e1 - t
      modpoly_add(e1, e1, t, m, iclen, pz);  // e1 = e1 + t
      e1 += oclen;
      e2 += oclen;
      pow += inc;
   }
}
```

4.3 Asserting the Code's Correctness

Development included writing a lot of test code. Every major function has some *unit tests* following it. The unit tests usually contain fixed data to be processed by the function to be tested and compare its output to results that are known to be correct, since they were computed by other means: Python programs were used to compute the correct results for FFTs in polynomial quotient rings, a different method for multiplication was used to test DKSS multiplication, and sometimes the correct results were more or less obvious and could be hard-coded by hand.

Additionally, functions contain *assertions* (like C++'s `assert()`), which are assumptions that are written together with the (proper) code and are checked at run-time. Often, these are pre- and post-conditions of functions. Some `asserts` call functions that were solely written for use in assertions, like a test for primitivity of a root.

To have the best of both worlds, code can be compiled in *Debug* or *Release mode* with Visual Studio. Release builds have all `assert`s disabled and are compiled with optimizations for maximum speed, while Debug builds feature assertion checking, but code optimization is disabled to aid debugging. Test code is usually run in Debug mode, while benchmarks are run in Release mode.

Furthermore, after development of DKSS multiplication was completed, it was integrated into my framework of long integer routines that is maintained as a private project. This framework is used for primality testing of Mersenne numbers (numbers of the form $2^p - 1$). Of course, it can not compare to the *Great Internet Mersenne Prime Search* [GIMPS], the distributed effort to find new Mersenne prime numbers that is going on since 1996 and has found the last eleven record prime numbers.

Nevertheless, I have been checking Mersenne numbers for primality for over two years now and a database exists of the low 64 bits of the result (called the *residue*) for each Mersenne number. The primality test used for Mersenne numbers is the *Lucas-Lehmer test* [CP05, ch. 4.2.1]. It consists of a loop of a long integer square, a subtraction by 2 and a modular reduction. The nature of this test causes even single-bit errors to proliferate, so any error would most likely alter the residue as well. Since it is hard to test all code paths with unit tests this makes it a good way to test a multiplication routine.

As a *system test* DKSS multiplication was used in Mersenne number primality tests and its results were compared against existing results. The first 35 Mersenne primes (the largest being $2^{1\,398\,269} - 1$) were correctly identified as such. Furthermore, all Mersenne numbers $2^p - 1$ with $p < 120\,607$ and various other sizes were tested and the residues matched.

4.4 Execution Time

Our main interest is to find out how fast DKSS multiplication is in comparison to other, well established algorithms. Except for small and medium lengths, Schönhage-Strassen multiplication was the fastest algorithm that used all-integer methods in practice so far. I compare both implementations `DKSS_MUL` and `SMUL` to one another.

Figure 19 shows graphs of `DKSS_MUL` and `SMUL` execution time (and Figure 24 shows some of the raw data). The cyan-colored and the magenta-colored graph show execution time if p was not rounded up to the next multiple of the word size, and if p was in fact rounded up, respectively (cf. Section 4.1). It is always faster to use a rounded up p than to use the "original" value.

As can be seen clearly, `DKSS_MUL` is much slower (about 30 times) than `SMUL` (printed in green) over the whole range of tested input lengths. From this graph it is hard to see if `DKSS_MUL` is gaining on `SMUL`. Section 4.8 discusses the quotient of run-times and the location of a crossover point in detail.

The stair-like graph stems from the fact that execution time almost totally depends on the FFT length $2M$ and the size of elements of $\mathcal{R} = \mathcal{P}/(\alpha^m + 1)$ with $\mathcal{P} = \mathbb{Z}/p^z\mathbb{Z}$. Since both M and m are powers of 2, many different input lengths lead to the same set of parameters.

The graph shows that execution time is almost the same for the beginning and the end of each step of the stair. The only part that depends directly on N is the encoding of the input numbers

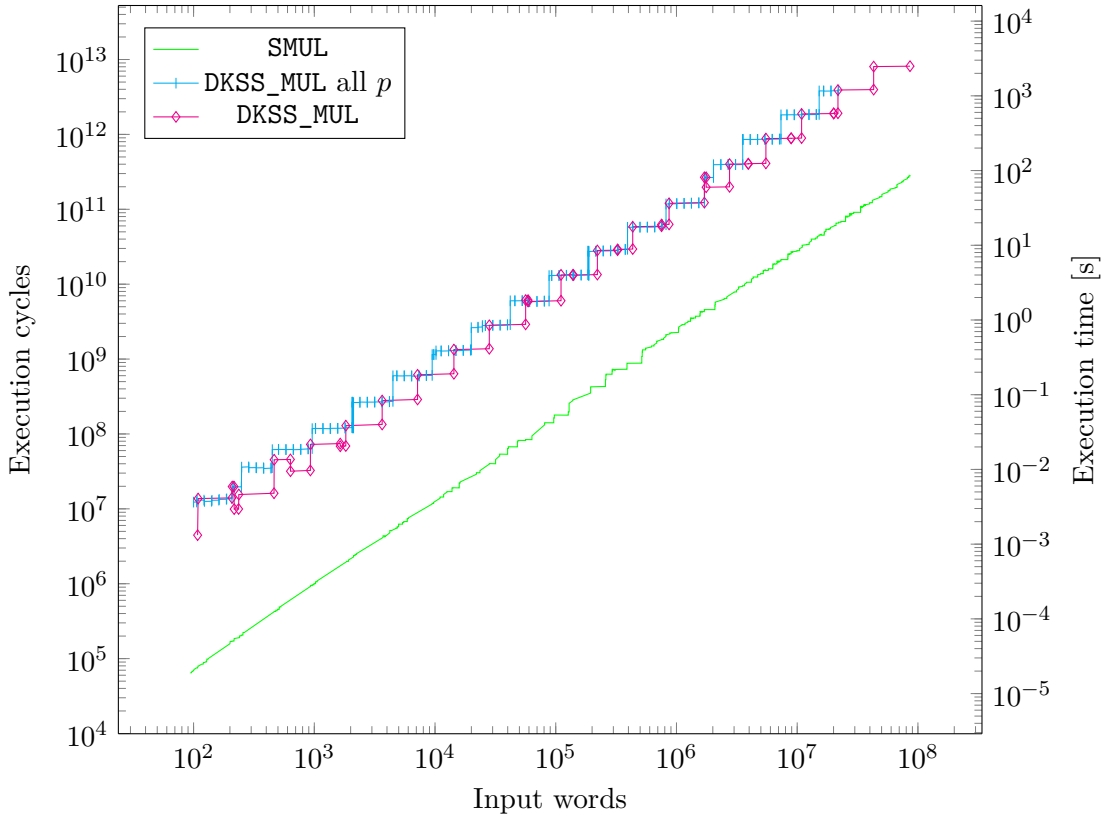

FIGURE 19: Execution time of `DKSS_MUL`

and decoding into the resulting product. But the time needed to do the FFT clearly dominates overall execution time.

In contrast to `DKSS_MUL`, the `SMUL` execution time graph is much smoother. In fact, it is reproduced without marks that would otherwise only obscure the graph, because there are a total of 12 969 data points available, of which 465 representative points are shown.

Obviously, `DKSS_MUL` parameter selection could be improved, since sometimes larger input numbers lead to faster execution times. Either, more research on parameter selection or a calibration process should smooth this out.

4.5 Memory Requirements

`DKSS_MUL` memory requirements are dominated by three times the size of the polynomials: input $a(x)$ and $b(x) \in \mathcal{R}[x]$ and the $\bar{a}_v(x)$. The result $c(x)$ requires no further memory, since storage of one of the input polynomials can be reused. An improved implementation could save the $\bar{a}_v(x)$ directly back into the polynomial without need for temporary storage, thus saving one third of memory requirements. To accomplish that a fast matrix transposition is needed, which in itself is not trivial (cf. [Knu97a, exercise 1.3.3-12]).

The polynomials each have $2M$ coefficients in $\mathcal{R} = \mathcal{P}[\alpha]/(\alpha^m + 1)$, where $\mathcal{P} = \mathbb{Z}/p^z\mathbb{Z}$. Hence, each polynomial needs $2Mm\lceil \log p^z \rceil$ bits. With $M \approx N/\log^2 N$, $m \approx \log N$ and $p^z \approx$

Input length (words)	`DKSS_MUL` memory (bytes)	`DKSS_MUL` blow-up	`SMUL` memory (bytes)	`SMUL` blow-up	Q
3648	803 584	27.54	251 848	8.63	3.19
7168	1 623 040	28.30	501 704	8.75	3.24
14 336	3 228 672	28.15	962 728	8.39	3.35
28 160	6 439 936	28.59	1 855 528	8.24	3.47
56 320	12 862 464	28.55	3 693 672	8.20	3.48
110 592	25 707 520	29.06	7 240 136	8.18	3.55
221 184	51 422 464	29.06	14 331 384	8.10	3.59
434 176	102 819 072	29.60	28 372 232	8.17	3.62
868 352	205 612 288	29.60	56 716 552	8.16	3.63
1 703 936	406 915 072	29.85	111 269 224	8.16	3.66
2 752 512	616 798 080	28.01	178 538 880	8.11	3.45
5 505 024	1 233 557 376	28.01	361 056 896	8.20	3.42
10 878 976	2 467 113 216	28.35	705 184 592	8.10	3.50
21 757 952	4 934 174 976	28.35	1 477 143 728	8.49	3.34
42 991 616	9 765 277 440	28.39	2 819 507 024	8.20	3.46
85 983 232	19 530 403 584	28.39	5 638 486 864	8.20	3.46

FIGURE 20: Memory requirements of `DKSS_MUL` and `SMUL`

$\frac{1}{2}N^5/\log N$ (see (4.1)) that results in

$$\begin{aligned} 2Mm\lceil\log p^z\rceil &\approx 2N/\log^2 N \cdot \log N \cdot \log(\frac{1}{2}N^5/\log N) \\ &= 2N/\log N \cdot (-1 + 5\log N - \log\log N) \\ &\approx 10N. \end{aligned}$$

The listing of function `dkss_fft()` in Section 4.2 shows that more memory, namely another $(2m+1)\cdot\lceil\log p^z\rceil \approx 10\log^2 N$ bits, is allocated, but compared to $10N$ bits for each polynomial that is of no big consequence. The same applies to the $2M/2m$ precomputed powers of ρ, each with a length of $m\lceil\log p^z\rceil$ bits. Together, they only need $2M/2m \cdot m\lceil\log p^z\rceil = M\lceil\log p^z\rceil$ bits, that is, a $2m$-th part of the memory of one polynomial. Hence, if both input numbers have N bits, total memory needed by `DKSS_MUL` is

$$M_{\texttt{DKSS\_MUL}}(N) \approx 30N \text{ bits.}$$

Let us now compare the memory requirements of `DKSS_MUL` to `SMUL`. According to (2.32), $M_{\texttt{SMUL}}(N') = 4N'$ bits. I wrote "N'", since in Chapter 2.9 "N" describes the length of the *product*, hence $N' = 2N$ to adjust notation to this chapter. Ergo, the approximate amount of temporary memory for `SMUL` is $M_{\texttt{SMUL}}(N) = 4N' = 8N$ bits.

Figure 20 shows an overview of actual memory consumption for selected input sizes for both `DKSS_MUL` and `SMUL`. The lengths chosen are the most favorable lengths for `DKSS_MUL`. At those lengths, the coefficients of the polynomials in `DKSS_MUL` are fully filled with bits from input numbers a and b (as much as possible, as the upper half of each polynomial still has to be zero to leave room for the product). Increasing the lengths by one would lead to the least favorable lengths that need about double the memory for almost the same input length.

The column "`DKSS_MUL` blow-up" shows the quotient of `DKSS_MUL` memory requirements and the size of *one* input factor in bytes. The column "`SMUL` blow-up" shows the same quotient for `SMUL`. The column "Q" shows the quotient of `DKSS_MUL` and `SMUL` memory requirements. Column "`DKSS_MUL` blow-up" nicely fits the approximated memory of $30N$ as well as column "`SMUL` blow-up" supports the approximated memory requirements of $8N$.

4.6 Source Code Size

Given the description of the DKSS algorithm in Chapter 3, the implementation is relatively straight-forward. About one third of the newly written code is needed for performing polynomial arithmetic: addition, subtraction, comparison, cyclic shifting and output and furthermore, using Kronecker-Schönhage substitution, multiplication, squaring and exponentiation. The other two thirds are taken up by the core DKSS routines, code to compute the primes p and other supporting code.

Underlying the DKSS code are routines that had to be written, but are not otherwise mentioned here, since they are not an immediate part of DKSS multiplication, like: factoring of long integers into primes and Lucas primality test [CP05, sec. 4.1] (for the computation of primes p for rings $\mathcal{P}$), extended Euclidean algorithm (to compute modular inverses in Hensel lifting and Lagrange interpolation), a C++ class for long numbers (to handle non-time-critical calculations easily), a faster division with remainder (see [Knu97b, ch. 4.3.1, p. 272] and [BZ11, ch. 1.4.1]). Other code that was used had already been written before: basic arithmetic, benchmarking code for speed tests, the Lucas-Lehmer test for primality for Mersenne numbers and a database of Mersenne number primality test results.

To give an idea about the size of the source code of DKSS multiplication, the following table shows the counts of lines of code. The second column ("Total source lines") contains the count including test and debug code, assertions, comments and empty lines, while the third column excludes those and only counts lines of code that actually do work in a production version ("Pure code lines"). The big difference in numbers is mostly because of test code. The above mentioned underlying routines are not included in the counts.

Description	Total source lines	Pure code lines
Polynomial arithmetic	958	295
Core DKSS multiplication	1374	336
Precomputation of primes p	139	86
Other supporting code	279	157
Total program code	2750	874
Table of precomputed primes p	1707	1705
Total	4457	2579

"Table of precomputed primes p" contains an array of 1703 prime numbers of the form $h \cdot 2^n + 1$ for each bit length from 2 to 1704, with the smallest odd h. Data from this array is needed for `DKSS_MUL`, but it doesn't really qualify as code, because it's only a list of constants. Since only values for p are used that are a multiple of 64 bits long and input numbers are limited by the 64-bit address space of the CPU, a list with 6 values for p would have done as well.

Compare this to the line counts of the implementation of Schönhage-Strassen multiplication:

Description	Total source lines	Pure code lines
Core `SMUL` multiplication	805	323
Fast cyclic shifts	518	253
Other supporting code	414	237
Total	1737	813

The row "Fast cyclic shifts" shows a special feature of the `SMUL` implementation: I went to great lengths to write fast cyclic shift code that takes advantage of different shift counts (like word- or byte-aligned). The original function for cyclic shifts had only 4 lines!

4.7 Profiling

To get a feeling for which parts of `DKSS_MUL` use up the most computing time, I did some *profiling* of the code. Visual Studio's built-in profiling did not perform very accurately and I had some technical difficulties. So instead I used a small self-made solution: I timed the execution of certain code parts manually.

This is not a thorough investigation, but just serves to gain a better understanding where *hot spots* of execution lie. Thus, I have chosen just five different input lengths for measurement.

In a first run, I measured the execution times for FFT setup (precomputation of ρ and its powers), the time needed for all three FFTs, pointwise multiplications and encode/decode/normalization of the result.

Input length (words)	FFT setup	`dkss_fft()`	Pointwise multiplications	En/decode & normalize
3648	18.00 %	58.60 %	16.55 %	6.85 %
28 160	2.26 %	79.46 %	12.88 %	5.40 %
221 184	0.66 %	84.40 %	10.53 %	4.41 %
10 878 976	0.39 %	88.56 %	8.19 %	2.86 %
42 991 616	0.27 %	87.71 %	9.32 %	2.70 %

FIGURE 21: Profiling percentages for `DKSS_MUL`

Figure 21 shows the results. I only present percentages of execution time. From this table several conclusions can be drawn:

- Computation of ρ and its powers, something which has to be done before the FFT starts, takes a diminishing share of time as the input gets longer. When numbers are in the millions of words long, it doesn't carry any significant weight in the overall run-time. This was to be expected.
- The same holds in principle for encoding, decoding and normalizing of the polynomials. It's more expensive than computing ρ and its powers, but with a decreasing share of the total cost. This too, was to be expected.

- Even the pointwise multiplications seem to be getting less prominent in the overall cost. Maybe this shows that parameters could be selected better? More research is needed here.
- The one part which is taking a growing share of the total cost is the DKSS FFT itself. I cannot assess from this data whether the share will be ever growing or reaches a plateau. Still, most of the execution time is spent here, so this is why we look more closely into its run-time.

In Figure 22 we see the percentages of execution time that are needed by the constituent parts of the DKSS FFT. It is performed by computing inner DFTs, bad multiplications and outer DFTs, which for their part are calculated by recursively calling the FFT routine and therefore again calculating inner DFTs and bad multiplications. The respective columns contain the execution time summed up over all levels of recursion. This table is normalized, so that total time of `dkss_fft()` is 100 %.

Input length (words)	Inner FFT	Bad multiplications	Rest
3648	22.24 %	76.09 %	1.67 %
28 160	16.98 %	80.90 %	2.12 %
221 184	16.20 %	81.23 %	2.57 %
10 878 976	10.21 %	87.66 %	2.13 %
42 991 616	9.50 %	89.36 %	1.14 %

FIGURE 22: Profiling percentages for `dkss_fft()`

The column titled "Rest" contains some call overhead, the copying of $a_{j\mu+\ell}$ into e_ℓ and the copy back of the $\bar{a}_v$ coefficients into the a array. I suspected that cache thrashing would slow this process down a lot, but these results show that this is not the case.

From this more specific analysis we learn that most of the time in `dkss_fft()` is used up by bad multiplications and their share is growing. That sure is a hot spot. So we will have a look into bad multiplications, which are multiplications of two arbitrary elements of $\mathcal{R}$.

Figure 23 shows a breakdown of execution time for multiplications of elements of $\mathcal{R}$. Multiplications are done by Kronecker-Schönhage substitution: encode polynomials as integers, multiply the integers, decode them back to polynomials, perform the "wrap around", that is, the modulo $(\alpha^m + 1)$ operation, and perform the modulo p^z operation on the inner coefficients. Again, total time of bad multiplications was normalized to 100 %.

Input length (words)	m	Words per inner coefficient	Integer multiplication	Modular reduction	Rest
3648	16	2	47.33 %	38.92 %	13.75 %
28 160	16	2	46.72 %	39.69 %	13.59 %
221 184	16	2	46.57 %	39.80 %	13.63 %
10 878 976	16	3	57.37 %	33.14 %	9.49 %
42 991 616	32	3	66.48 %	26.67 %	6.84 %

FIGURE 23: Profiling percentages for bad multiplications

Since Kronecker-Schönhage substitution depends on $\mathcal{R}$, it only depends on parameters m and p^z, but not M nor u. The first three rows have the same values for m and p^z, so it fits the theory well that the percentages are more or less the same.

Time needed for modular reduction is not negligible and a better means than modulo division might save some time here (*Fast mod operation for Proth moduli*, [CP05, p. 457]). But the trend seems to be that for growing lengths the share of execution time needed for modular reductions is shrinking.

In column "Rest" the times for encoding and decoding between polynomials and integers are lumped together. This seems to be quite slow and a more careful implementation could speed it up, but again, that percentage will only drop as input numbers get longer.

From this profiling analysis we have learned that bad multiplications are really bad! Up to 90 % of execution time is spent there and its share is growing. In order to reduce overall execution time, we should reduce the number of bad multiplications and/or make them cheaper. Maybe better parameter selection could reduce execution time here, which is left open for future research.

4.8 Gazing into the Crystal Ball

One goal of this thesis is to compare the speed of a `DKSS_MUL` implementation with an `SMUL` implementation. As was described in Section 4.4, `SMUL` is still much faster for the lengths tested.

In addition, it would be interesting to estimate the input length where `DKSS_MUL` starts to be faster than `SMUL`. To do that, we look again at the most favorable lengths for `DKSS_MUL`, that is, the lower right points of the steps in Figure 19, where the execution time graph for `DKSS_MUL` is nearest to the `SMUL` graph. Figure 24 lists execution times at those points and the quotient of these times. Figure 25 shows a graph of the quotient of execution times vs. input length.

Length (words)	`DKSS_MUL` time (cycles)	`DKSS_MUL` (min:sec)	`SMUL` time (cycles)	`SMUL` (min:sec)	Quotient
3648	133 948 102	0:00.039	4 149 866	0:00.001	32.28
7168	288 821 718	0:00.085	8 884 604	0:00.003	32.51
14 336	636 214 972	0:00.187	19 131 288	0:00.006	33.26
28 160	1 373 645 624	0:00.404	39 547 108	0:00.012	34.73
56 320	2 908 271 180	0:00.855	81 912 772	0:00.024	35.50
110 592	6 013 189 608	0:01.769	179 448 020	0:00.053	33.51
221 184	13 430 829 526	0:03.950	425 460 492	0:00.125	31.57
434 176	29 461 464 342	0:08.665	882 781 300	0:00.260	33.37
868 352	62 917 787 338	0:18.505	2 167 722 116	0:00.638	29.02
1 703 936	122 680 187 946	0:36.082	4 576 352 552	0:01.346	26.81
2 752 512	199 424 397 176	0:58.654	7 495 493 476	0:02.205	26.61
5 505 024	410 390 455 672	2:00.703	15 269 441 152	0:04.491	26.88
10 878 976	892 949 727 060	4:22.632	31 013 681 856	0:09.122	28.79
21 757 952	1 917 703 330 120	9:24.030	65 485 660 216	0:19.260	29.28
42 991 616	3 965 210 546 518	19:26.238	132 248 494 436	0:38.897	29.98
85 983 232	8 145 120 758 260	39:55.624	288 089 862 672	1:24.732	28.27

FIGURE 24: Execution times of `DKSS_MUL` and `SMUL`

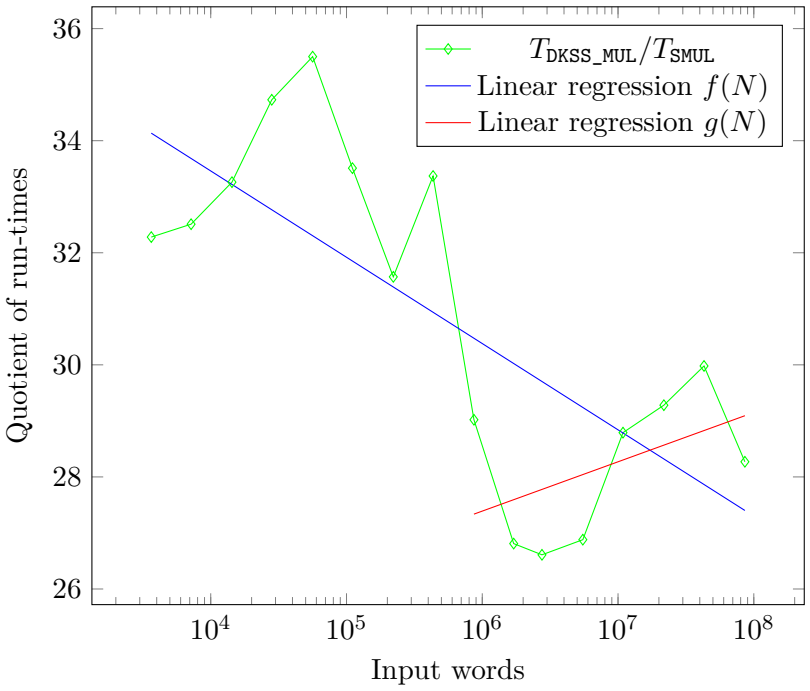

FIGURE 25: Quotient of `DKSS_MUL` and `SMUL` run-times vs. input length

At first sight, there is an apparent trend in the quotient of execution times. Looking at Figure 25 we might, as a first approximation, assume a linear relationship between $\log_{10} N$ and the quotient of execution times. Linear regression with a least squares estimation leads to the line $f(N) = -1.54 \cdot \log_{10} N + 39.62$, which has a correlation coefficient of -0.723. Solving $f(N) = 1$ leads to $N \approx 10^{25} \approx 2^{83}$ bits.

On the other hand, analysis of `SMUL` execution times in Section 2.9.5 showed that `SMUL` reaches its "final" speed only above input lengths of about 512 Kwords (cf. Figure 13). So it seems that Figure 25 not so much shows the speed-up through improved speed of `DKSS_MUL`, but the slow-down of `SMUL` because of diminishing positive effects of caching. If we do linear regression with data points starting at input length 512 Kwords only, we get $g(N) = 0.88 \cdot \log_{10} N + 22.11$, that is, the quotient would be growing! Obviously, this type of analysis is not very reliable.

As we did for `SMUL` in Section 2.9.5, we can use the measured data points to try to model the run-time of `DKSS_MUL`. Writing (3.19) with an explicit constant we get

$$T_\delta(N) \le N \cdot \log N \cdot 2^{\delta \cdot \log^* N}. \tag{4.2}$$

Calculating the constant δ from each data point and plotting all of them gives the graph in Figure 26, with average $\delta \approx 1.2727$. In contrast to Figure 13, no effect of caching is apparent. We only have few data points, so this model of run-time is not very resilient. Yet, the modeled run-time matches the measured values ± 10 % and even within ± 5 % for input lengths $\ge 28\,160$ words. With the few data points we have, it seems to be the best we can do.

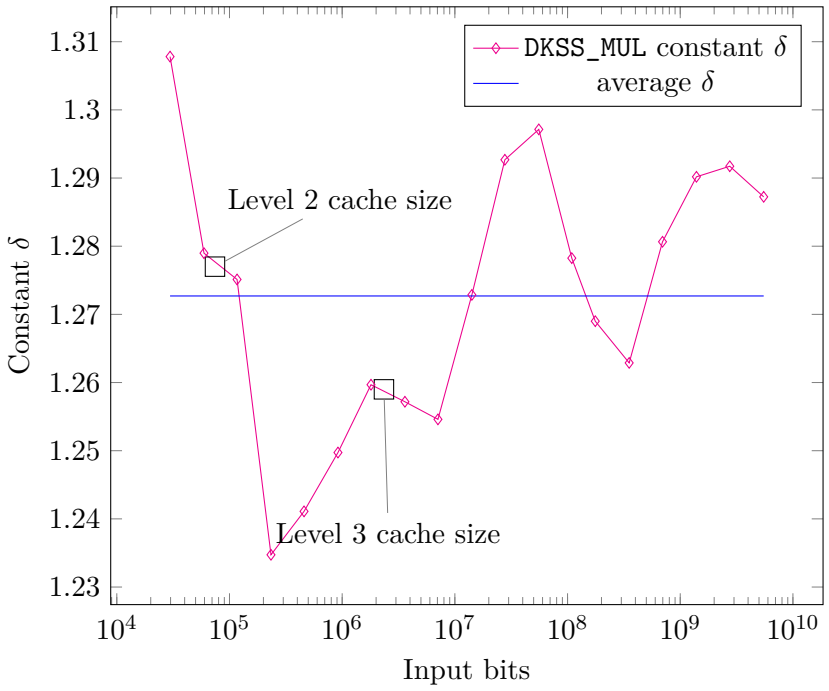

FIGURE 26: `DKSS_MUL` constant δ

Taking (2.33) and (4.2) we can solve

$$\begin{aligned} T_\delta(N) &\leq T_\sigma(N) \\ N \log N \cdot 2^{\delta \log^* N} &\leq \sigma N \log N \cdot \log\log N \\ 2^{\delta \log^* N} &\leq \sigma \log\log N \\ \delta \log^* N &\leq \log \sigma + \log\log\log N. \end{aligned}$$

For large N we substitute $\nu \coloneqq \log\log N$ and with (3.16) get

$$\begin{aligned} \delta(\log^*(\log\log N) + 2) &\leq \log\sigma + \log\log\log N \\ \delta(\log^* \nu + 2) &\leq \log\sigma + \log\nu. \end{aligned} \tag{4.3}$$

Solving (4.3) numerically yields the enormous solution of $\nu \geq 498$ and hence $N \geq 10^{10^{149}}$ bits! An optimistic estimation of the number of bits for computer memory available in this universe is 10^{100}. So this crossover point is *orders of orders* of magnitude higher than any machine could hold that anyone could ever build.

Even if `DKSS_MUL` was only about 2 times slower than `SMUL`, the crossover point would still be at $N \approx 10^{300}$ bits and thus unreachable.

Chapter 5

Conclusion

De, Kurur, Saha and Saptharishi describe a new procedure to multiply very large integers efficiently (cf. Chapter 3, implemented as `DKSS_MUL`). The currently widely used all-integer multiplication algorithm for large numbers is by Schönhage and Strassen [SS71] (my implementation is called `SMUL`, cf. Section 2.9). The run-time of `DKSS_MUL` is in a better complexity class than that of `SMUL`, meaning that if input numbers are long enough, `DKSS_MUL` will be faster than `SMUL`. Both algorithms were implemented and their run-time (Section 4.4) and memory consumption (Section 4.5) were compared (on a PC with 32 GB memory and a 3.4 GHz processor).

The results indicate that Schönhage and Strassen's multiplication algorithm is the better choice for a variety of reasons:

1. `SMUL` is faster than `DKSS_MUL`.

   Benchmarks show that `SMUL` is still about 26 to 36 times faster than `DKSS_MUL` (Section 4.4 and especially Figures 19 and 25). The estimate of the input length at which `DKSS_MUL` is faster than `SMUL` (Section 4.8) is $N \geq 10^{10^{149}}$ bits (which is larger than googolplex), but even if `SMUL` was only 2 times faster than `DKSS_MUL`, the crossover point would be so large that it could never be reached.

2. `SMUL` requires less memory than `DKSS_MUL`.

   If both input numbers are N bits long, `DKSS_MUL` requires about $30N$ bits of temporary memory, where `SMUL` requires only about $8N$ bits (Sections 4.5 and 2.9.4). The memory requirements of `SMUL` can not be lowered significantly, but there is an obvious possibility to lower `DKSS_MUL` memory consumption to its lower limit of about $20N$ bits that was not implemented.

3. `SMUL` is easier to implement than `DKSS_MUL`.

   A simple implementation of `SMUL` needs about 550 lines of C++ code, where `DKSS_MUL` requires about 900 lines plus at least 6 lines of constants and more supporting routines, see Section 4.6. An improved and faster version of `SMUL` requires about 800 lines of code.

It should be mentioned here that the `SMUL` implementation is better optimized than `DKSS_MUL`. The reason for that is that Schönhage-Strassen multiplication is now studied and in wide use for many years and its details are well understood. I have spent considerable time to improve its implementation. In contrast, DKSS multiplication is still quite young and to my knowledge

this is the first implementation of it. Section 5.1 describes several possible improvements to `DKSS_MUL` that could be realized. Still, in my appraisal none of them has the potential to speed up `DKSS_MUL` so much that it becomes faster than `SMUL` in the range of input lengths that was examined here or even in ranges that might be accessible in the future.

5.1 Outlook

In the course of writing, I encountered several possible areas for improvement. I list them here and try to assess their potential to improve run-time.

- Find optimum values of parameters M, m, u and p^z for any given N.

  Figure 19 still shows some areas where longer input numbers lead to shorter execution times. Furthermore, Section 4.7 shows some developments in percentages of run-times that could suggest that a better choice of parameters is possible. More research is needed to understand how to choose the fastest set of parameters.

- Cache computation of ρ and its powers.

  This is an obvious possibility to save execution time, but it cannot save a great share when numbers get longer. Figure 21 shows how the percentage of execution time of "FFT setup" diminishes as numbers get longer. This has no potential to lower the crossover point.

- Add support for "sparse integers" in the underlying multiplication.

  `DKSS_MUL` reduces multiplication of long integers to multiplications in $\mathcal{R}$, a polynomial ring. When it comes to multiplication of two elements of $\mathcal{R}$, they are again converted to integers (via Kronecker-Schönhage substitution, see Section 3.2.6) and have to be padded with zeros. About half of the words of each factor are zero and a future multiplication routine could exploit that. Profiling in Section 4.7 showed that up to 85 % of execution time is spent with multiplication of elements of $\mathcal{R}$ and a rising percentage of that is used by the underlying integer multiplication. I optimistically estimate the potential of this idea to speed up `DKSS_MUL` to be almost a factor of 2.

- Count the number of non-zero coefficients in Kronecker-Schönhage substitution.

  We have to pad the polynomial coefficients for Kronecker-Schönhage substitution (cf. Section 3.2.6) with zeros, partly because multiple coefficient products are summed up and we must prevent that sum from overflowing. By counting the number of non-zero coefficients prior to multiplying them, we could upper bound the number of products. I estimate one or two bits of padding per coefficient product could be saved, but since coefficients are themselves at least 64 bits long, their product is at least 128 bits, so the potential saving can be no more than about 1–2 % and shrinks when numbers and thus coefficients get longer.

- Implement `dkss_fft()` with less extra memory but matrix transposition instead.

  This is definitely an improvement that should be implemented, because it brings down the memory requirements from about $30N$ bits to about $20N$ bits (cf. Section 4.5). Yet, from the numbers obtained by profiling, I estimate the potential saving in run-time to be only a few percent at best. Furthermore, it seems that efficient matrix transposition by itself is non-trivial.

- Exploit the build of Proth prime numbers p.

  The modulus of $\mathcal{P}$ is a prime number of the form $h \cdot 2M + 1$, where h is a small positive odd integer and M is a power of 2. Maybe modular reductions can be sped up by the technique listed in [CP05, p. 457]. This has the potential to save a great part of the cost of modular reductions, which showed to cost about 22 % of run-time in profiling.

If all potential savings listed above could be achieved, this would speed up `DKSS_MUL` by a factor of about 2.5. Not included in this factor is a better parameter selection. But even if that and other, yet unthought-of, improvements lead to another speed-up by a factor of 2, `DKSS_MUL` would still be at least 4.8 times slower than `SMUL` and need about 2.5 times more memory. As explained on page 63, even then the crossover point could never be reached.

Appendix A

Technicalities

Tests and benchmarks were run on a machine with an Intel Core i7-3770 processor (Ivy Bridge microarchitecture) with 3.40 GHz clock rate. Hyper-threading, enhanced SpeedStep and Turbo Boost were disabled to enhance accuracy of timings. The mainboard is an ASUS P8Z77-V with 32 GB PC-1600 dual channel DDR3 memory.

The CPU has four cores, of which only one core was used while benchmarking. That is, the other cores were not switched off, but no other CPU-intensive process was running, except for the operating system itself. To improve cache performance, the process affinity was fixed to processor 2, which seems to get less interrupt and DPC load than processor 0.

The CPU has level 1 caches per core of both 32 KB for data and 32 KB for instructions, unified level 2 caches of 256 KB per core and a unified level 3 cache of 8 MB for all cores. Caches lines are 64 bytes long and all caches are 8-way set associate, except the level 3 cache, which is 16-way set associative.

The operating system used was Windows 7 Ultimate with Service Pack 1 in 64-bit mode.

For benchmarking, the priority class of the process was set to the highest non-realtime value, that is, `HIGH_PRIORITY_CLASS`. The thread priority was also the highest non-realtime value, `THREAD_PRIORITY_HIGHEST`. Together, that results in a base priority level of 13.

Timings were taken by use of Windows' `QueryThreadCycleTime()` function that counts only CPU cycles spent by the thread in question. It queries the CPU's *Time Stamp Counter* (TSC) and its resolution is extremely good: even though the CPU instruction `RDTSC` is not serializing (so some machine language instructions might be executed out-of-order), the accuracy should be of the order of a 100 cycles at worst, most likely better.

As development environment Microsoft's Visual Studio 2012, v11.0.61030.00 Update 4 was used which includes the C++ compiler v17.00.61030. Code was compiled with options `/Ox` (full optimization), `/Ob2` (expand any suitable inline function), `/Oi` (enable intrinsic functions), `/Ot` (favor fast code) and `/GL` (whole program optimization).

Bibliography

[AHU74] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.

[BZ06] Marco Bodrato and Alberto Zanoni. What about Toom-Cook Matrices Optimality? http://bodrato.it/papers/WhatAboutToomCookMatricesOptimality.pdf, October 2006.

[BZ11] Richard P. Brent and Paul Zimmermann. *Modern Computer Arithmetic*. Cambridge University Press, 2011.

[CB93] Michael Clausen and Ulrich Baum. *Fast Fourier Transforms*. B.I.-Wissenschaftsverlag, 1993.

[CLRS09] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, 3rd edition, 2009.

[Coo66] Stephen A. Cook. *On the Minimum Computation Time of Functions*. PhD thesis, Harvard University, 1966.

[CP05] Richard Crandall and Carl Pomerance. *Prime numbers: A Computational Perspective*. Springer, 2nd edition, 2005.

[CT65] James W. Cooley and John W. Tukey. An algorithm for the machine calculation of complex Fourier series. *Math. Comput.*, 19:297–301, 1965.

[DKSS08] Anindya De, Piyush P. Kurur, Chandan Saha, and Ramprasad Saptharishi. Fast Integer Multiplication Using Modular Arithmetic. In *ACM Symposium on Theory of Computing*, pages 499–506, 2008.

[DKSS13] Anindya De, Piyush P. Kurur, Chandan Saha, and Ramprasad Saptharishi. Fast Integer Multiplication Using Modular Arithmetic. *SIAM Journal on Compution*, 42(2):685–699, 2013.

[DV90] P. Duhamel and M. Vetterli. Fast Fourier Transforms: A Tutorial Review and a State of the Art. *Signal Processing*, 19:250 – 299, 1990.

[Fü07] Martin Fürer. Faster Integer Multiplication. In *Proceedings of the 39th ACM Symposium on Theory of Computing*, pages 57–66, 2007.

[Fü09] Martin Fürer. Faster Integer Multiplication. *SIAM Journal on Compution*, 39(3):979–1005, 2009.

[Fis11] Gerd Fischer. *Lehrbuch der Algebra*. Vieweg & Teubner, 2nd edition, 2011.

[GG13] Joachim von zur Gathen and Jürgen Gerhard. *Modern Computer Algebra*. Cambridge University Press, 3rd edition, 2013.

[GIMPS] George Woltman, Scott Kurowski, et al. Great Internet Mersenne Prime Search. http://www.mersenne.org/.

[GKZ07] Pierrick Gaudry, Alexander Kruppa, and Paul Zimmermann. A GMP-based Implementation of Schönhage-Strassen's Large Integer Multiplication Algorithm. In *International Symposium on Symbolic and Algebraic Computation*, 2007.

[GLTZ10] Kaveh Ghazi, Vincent Lefèvre, Philippe Théveny, and Paul Zimmermann. Why and How to Use Arbitrary Precision. *Computing in Science and Engineering*, 12(3):62–65, May–June 2010.

[GMP14] Torbjörn Granlund and the GMP development team. The GNU Multiple Precision Arithmetic Library Manual. https://gmplib.org/gmp-man-6.0.0a.pdf, March 2014.

[HJB85] Michael T. Heideman, Don H. Johnson, and C. Sidney Burrus. Gauss and the History of the fast Fourier transform. *Archive for History of Exact Sciences*, 34(3):265–277, 1985.

[Kar95] A. A. Karatsuba. The Complexity of Computations. *Proceedings of the Steklov Institute of Mathematics*, 211:169–183, 1995.

[Knu97a] Donald E. Knuth. *The Art of Computer Programming, Volume 1: Fundamental Algorithms*. Addison-Wesley, 3rd edition, 1997.

[Knu97b] Donald E. Knuth. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*. Addison-Wesley, 3rd edition, 1997.

[KO63] A. Karatsuba and Yu. Ofman. Multiplication of multidigit numbers on automata. *Soviet Physics – Doklady*, 7:595–596, 1963.

[Lin44a] U. V. Linnik. On the least prime in an arithmetic progression, I. The basic theorem. *Mat. Sbornik N. S.*, 15(57):139–178, 1944.

[Lin44b] U. V. Linnik. On the least prime in an arithmetic progression, II. The Deuring-Heilbronn phenomenon. *Mat. Sbornik N. S.*, 15(57):347–368, 1944.

[MPIR12] Torbjorn Granlund, William Hart, and the GMP and MPIR Teams. The Multiple Precision Integers and Rationals Library. http://www.mpir.org/mpir-2.6.0.pdf, November 2012.

[NZM91] Ivan Niven, Herbert S. Zuckerman, and Hugh L. Montgomery. *An Introduction to the Theory of Numbers*. John Wiley & Sons, 5th edition, 1991.

[Sch] Arnold Schönhage. Turing Processing, Turing Processor, Turing Programs. http://www.iai.uni-bonn.de/~schoe/tp/TPpage.html.

[Sch82] Arnold Schönhage. Asymptotically fast algorithms for the numerical multiplication and division of polynomials with complex coefficients. In Jacques Calmet, editor, *EUROCAM '82: European Computer Algebra Conference*, volume 144, pages 3–15, 1982.

[Sed92] Robert Sedgewick. *Algorithms in C++*. Addison-Wesley, 1992.

[SGV94] Arnold Schönhage, Andreas F. W. Grotefeld, and Ekkehart Vetter. *Fast Algorithms: a multitape Turing machine implementation*. B.I.-Wissenschaftsverlag, 1994.

[SS71] A. Schönhage and V. Strassen. Schnelle Multiplikation großer Zahlen. *Computing*, 7:281–292, 1971.

[Str01] Gilbert Strang. Lecture 22: Fourier expansions and convolution. http://videolectures.net/mit18085f07_strang_lec22, April 2001.

[Too63] A. L. Toom. The Complexity of a Scheme of Functional Elements Realizing the Multiplication of Integers. *Soviet Mathematics – Doklady*, 3:714–716, 1963.

[War02] Henry S. Warren, Jr. *Hacker's Delight*. Addison-Wesley, 2002.

[Xyl11] Triantafyllos Xylouris. *Über die Nullstellen der Dirichletschen L-Funktionen und die kleinste Primzahl in einer arithmetischen Progression*. PhD thesis, Universität Bonn, 2011.

[YL00] Chee Yap and Chen Li. QuickMul: Practical FFT-based Integer Multiplication. http://www.cs.nyu.edu/exact/doc/qmul.ps, October 2000.

[Zur94] Dan Zuras. More On Squaring and Multiplying Large Integers. *IEEE Transactions on Computers*, 43(8):899–908, 1994.

Index