Chapter 11

Gauss and Karatsuba Multiplication

Copyright Oliver Serang, 2018
University of Montana Department of Computer Science

11.1 Multiplication of complex numbers

As we remember from grade school, addition is fairly easy, while multiplication is more difficult. This coincides with an algorithmic analysis of addition and grade-school multiplication: Adding two \( n \)-digit numbers performs \( n \) operations on digit pairs\(^1\) up to \( n \) carry operations, and is thus \( \in O(n) \). Grade-school multiplication, on the other hand, pairs each digit from one number with every digit from the other number. Thus, even before these products are totaled, we have \( n \times n \) operations, and we can see that the grade-school multiplication algorithm is \( \in \Omega(n^2) \).

Historically, this meant that scientists doing computations by hand would gladly perform a few additions instead of a multiplication between two large numbers. This was the case with the scientist Gauß\(^2\) while working with

---

\(^1\)Aligned so that the 10’s place of one number is adding to the 10’s place of the other number, \textit{etc}.

\(^2\)My great-great-great-\ldots grand advisor\(^3\)

\(^3\)Or Groß-groß-\ldots Doktorvater
complex numbers, Gauß frequently performed complex multiplications
\[(a + b \cdot j) \cdot (c + d \cdot j) = a \cdot c - b \cdot d + (a \cdot d + b \cdot c) \cdot j.\]

where \(j = \sqrt{-1}\). This complex product can, of course, be found using four real products, \(a \cdot c, b \cdot d, a \cdot d,\) and \(b \cdot c\). This is implemented in Listing 11.1.

Listing 11.1: Multiplying complex numbers using the naive approach, with four multiplications.

```python
import numpy
class Complex:
    def __init__(self, real, imag=0.0):
        self.real = real
        self.imag = imag

    def __add__(self, rhs):
        return Complex(self.real+rhs.real, self.imag+rhs.imag)

    def __sub__(self, rhs):
        return Complex(self.real-rhs.real, self.imag-rhs.imag)

    def __mul__(self, rhs):
        return Complex(self.real*rhs.real-self.imag*rhs.imag,
                        self.real*rhs.imag+self.imag*rhs.real)

    def __str__(self):
        result=''
        if numpy.fabs(self.real) > 0.0 and numpy.fabs(self.imag) > 0.0:
            return str(self.real) + '+' + str(self.imag) + 'j'
        if numpy.fabs(self.real) > 0.0:
            return str(self.real)
        if numpy.fabs(self.imag) > 0.0:
            return str(self.imag) + 'j'
        return '0'

x=Complex(1,2)
y=Complex(2,3)
print x, y, x*y
```

At first glance, it may appear that there is no way to reduce the number of multiplications: every term in the first complex number must be multiplied with every term in the second complex number; however, because we are...
11.1. MULTIPLICATION OF COMPLEX NUMBERS

Table 11.1: The four terms in a complex multiplication.

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b · j</th>
</tr>
</thead>
<tbody>
<tr>
<td>c</td>
<td>a · c</td>
<td>b · c · j</td>
</tr>
<tr>
<td>d · j</td>
<td>a · d · j</td>
<td>−b · d</td>
</tr>
</tbody>
</table>

working on a “ring” space\(^1\) we can also add values, multiply them, and then subtract out. In this manner, we could potentially multiply some merged values and then subtract others out in order to get the result we want.

For example, if we wanted to compute \((a + b) \cdot (c + d)\), we could do so with the naive approach (using four multiplications), but we could also do so by computing \(e = a + b\) and \(f = c + d\), and then computing \(e \cdot f\) (using two multiplications). In a more sophisticated use of this approach, we could compute \((a + b) \cdot c\), \((a + b) \cdot d\), and \((a + b) \cdot (c + d)\) by first computing \(e = (a + b) \cdot c\) and \(f = (a + b) \cdot d\), and then computing \((a + b) \cdot (c + d) = e + f\). These approaches only add, but do not subtract, and thus they do not yet need the ring property.

Let us consider the four products that we perform when doing our complex multiplication (Table 11.1). The results of the complex multiplication will be the sum of terms on the diagonal (these will form the real result) and the sum of terms positive diagonal (these will form the complex result). Consider what would happen if we compute \(e = (a + b \cdot j) \cdot c\) and \(f = (c + d \cdot j) \cdot a\). The value of \(e\) will compute the sum of the first row in Table 11.1, while \(f\) will compute the sum of the first column in Table 11.1; therefore, we can use the ring property to subtract and compute the difference of terms on the positive diagonal. This will remove the top left cell of Table 11.1, leaving behind only contributions from the top right and bottom left cells in the table:

\[
e - f = a \cdot c + b \cdot c \cdot j - c \cdot a - d \cdot a \cdot j
= b \cdot c \cdot j - d \cdot a \cdot j.
\]

However, consider that the imaginary result of the complex multiplication will be \(b \cdot c \cdot j + d \cdot a \cdot j\); for this reason, we instead use \(e = (a + b \cdot j) \cdot c\) and

\(^1\)We are working on a “ring” whenever we are working on a set of objects that support addition and also support its inverse operation, subtraction.
$f = (c - b \cdot j) \cdot a$. Thus we invert the sign of that second term:

$$e - f = a \cdot c + b \cdot c \cdot j - c \cdot a + d \cdot a \cdot j$$

$$= b \cdot c \cdot j + d \cdot a \cdot j.$$

An important next step is removing whether values are real or imaginary so that the multiplication can be done on merged values. Specifically, if we let $e = (a + b) \cdot c$ and $f = (c - d) \cdot a$, then we have

$$e - f = a \cdot c + b \cdot c - c \cdot a + d \cdot a$$

$$= b \cdot c + d \cdot a,$$

which is the coefficient of the imaginary term in the result (i.e., we need only multiply $e - f$ with $j$ and we get the imaginary term). In this manner, we reduce the total number of multiplications to two instead of four when computing $e$ and $f$.

Of course, this does not yet confer an advantage over the naive approach: we have used two multiplications (one each to get $e$ and $f$), and we could compute the imaginary result using the naive approach with two multiplications (and no subtraction). But consider that in computing $e$ and $f$ in this way, that we have already computed the sum of the first row, $e$. If we compute the sum of terms in the second column, we could perhaps do the same as what we did above, and use the sum of the first row and the sum of the second column to subtract out and remove the top right cell of Table 11.1. Given $e = (a + b) \cdot c$, we let $g = (c + d) \cdot b$ and proceed as we did above:

$$e - g = a \cdot c + b \cdot c - c \cdot b - d \cdot b$$

$$= a \cdot c - b \cdot d.$$

We can compute $e$, $f$, and $g$ in three multiplications, and thus we compute the real part of the complex multiplication as $e - f$ and the imaginary part of the complex multiplication as $e - g$. In this manner, we use only three multiplications, although we need to introduce some additions and subtractions. Gauß actually discovered and used this approach when performing complex multiplications on large numbers; the extra cost of the linear-time additions and subtractions (of which there are a constant number) was well worth avoiding a large grade-school multiplication between to large numbers. This is implemented in Listing 11.2.
Listing 11.2: Multiplying complex numbers using Gauß multiplication, with three multiplications instead of four.

```
import numpy

class Complex:
    def __init__(self, real, imag=0.0):
        self.real = real
        self.imag = imag

    def __add__(self, rhs):
        return Complex(self.real+rhs.real, self.imag+rhs.imag)

    def __sub__(self, rhs):
        return Complex(self.real-rhs.real, self.imag-rhs.imag)

    def __mul__(self, rhs):
        # (a+bi)*(c+di)
        # a*c a*d = a*(c+d) = x
        # b*c -b*d = b*(c-d) = z
        # =
        # c*(b-a)
        # =
        # y

        x = self.real*(rhs.real+rhs.imag)
        y = rhs.real*(self.imag-self.real)
        z = self.imag*(rhs.real-rhs.imag)

        return Complex(z-y, x+y)

    def __str__(self):
        result=''
        if numpy.fabs(self.real) > 0.0 and numpy.fabs(self.imag) > 0.0:
            return str(self.real) + '+' + str(self.imag) + 'j'
        if numpy.fabs(self.real) > 0.0:
            return str(self.real)
        if numpy.fabs(self.imag) > 0.0:
            return str(self.imag) + 'j'

        return '0'
```

11.2 Fast multiplication of long integers

As impressive as it is, Gauß’s trick does not make the multiplications between large integers any easier. The principal trouble of this is that Gauß’s complex multiplication trick does not recurse: it reduces a complex multiplication to three real multiplications (and some additions and subtractions).

In the mid 20th century, the famous scientist Kolmogorov (now famous for Kolmogorov complexity and the Kolmogorov-Smirnov test) conjectured that multiplication between two n-digit integers was \( \in \Omega(n^2) \), that essentially there was no method substantially better than the grade-school algorithm, wherein every digit of one number touches every digit of the other number. At place where he presented this conjecture, a young man named Karatsuba was in the audience. Karatsuba tried to attack this problem and disproved Kolmogorov’s conjecture, and in doing so, Karatsuba discovered the first algorithm for multiplying two n-digit numbers \( \in o(n^2) \).

The trick behind Karatsuba’s algorithm was essentially to use Gauß-algorithm, but instead of multiplying two complex numbers, Karatsuba used it to multiply two n-digit integers. In this manner, he reduced the product of two n-digit integers to smaller problems of the same type (whereas Gauß had reduced complex multiplication to real multiplication, a different problem).

Consider the product between two integers, \( x \) and \( y \). We can split \( x \) into its most-significant digits \( x_{\text{high}} \) and its less-significant digits \( x_{\text{low}} \). In this manner \( x = x_{\text{high}} \cdot 10^n + x_{\text{low}} \), where \( x \) has \( n \) decimal digits\(^5\):

\[
x \cdot y = (x_{\text{high}} \cdot 10^n + x_{\text{low}}) \cdot (y_{\text{high}} \cdot 10^n + y_{\text{low}})
= x_{\text{high}} \cdot y_{\text{high}} \cdot 10^{2n} + (x_{\text{high}} \cdot x_{\text{low}} + x_{\text{low}} \cdot x_{\text{high}}) \cdot 10^n + x_{\text{low}} \cdot y_{\text{low}}.
\]

Table 11.2 rewrites the table from the Gauß multiplication.

This will produce the runtime recurrence \( r(n) = 4r\left(\frac{n}{2}\right) + \Theta(n) \). Using the master theorem, we see that this is leaf-heavy with \( r(n) \in \Theta(n^{\log_2(4)}) =
\]

\(^5\)For convenience, assume that \( n \) is divisible by 2.
11.2. FAST MULTIPLICATION OF LONG INTEGERS

\[ x_{\text{high}} \cdot 10^n \]
\[ x_{\text{low}} \cdot y_{\text{high}} \cdot 10^n \]
\[ x_{\text{high}} \cdot y_{\text{high}} \cdot 10^n \]
\[ x_{\text{low}} \cdot y_{\text{low}} \]

Table 11.2: The four terms in a recursive integer multiplication.

\[ \Theta(n^2) \]; it realizes but does not improve over the grade-school multiplication method.

To compute the integer product, we need to compute \( x_{\text{high}} \cdot y_{\text{high}} \) (which will occur at the significance \( 10^n \)) and \( x_{\text{low}} \cdot y_{\text{low}} \) (which will occur at the significance \( 10^0 \)). The remaining two terms both have significance \( 10^{\frac{n}{2}} \). Unlike Gauss’ trick, there is no need for a sign change, which simplifies things. We simply compute three products: First, \( z_{\text{high}} = x_{\text{high}} \cdot y_{\text{high}} \) and \( z_{\text{low}} = x_{\text{low}} \cdot y_{\text{low}} \) compute the highest and lowest significances in the result. Lastly, we compute the sum of all terms in Table 11.2 (with their significances stripped away), \( z_{\text{total}} = (x_{\text{high}} + x_{\text{low}}) \cdot (y_{\text{high}} + y_{\text{low}}) \). The medium-significance terms of the result (which occur on the positive diagonal of Table 11.2) can be recovered by subtracting \( z_{\text{medium}} = z_{\text{total}} - z_{\text{high}} - z_{\text{low}} \).

Thus, the low-, medium-, and high-significance parts of \( z = x \cdot y \) can be computed with three multiplications. We reassemble these into \( z = z_{\text{high}} \cdot 10^n + z_{\text{medium}} \cdot 10^{\frac{n}{2}} + z_{\text{low}} \). Karatsuba’s method reduces an \( n \)-digit multiplication to three \( \frac{n}{2} \)-digit multiplications and a constant number of additions and subtractions; therefore, we have \( r(n) = 3r(\frac{n}{2}) + \Theta(n) \), which is \( \in \Theta(n\log_2(3)) = \Theta(n^{1.585...}) \) using the leaf-heavy case of the master theorem. An implementation of an arbitrary-precision integer class (with implementations of both naive \( \Omega(n^2) \) and Karatsuba \( \Theta(n^{1.585...}) \) multiplication) is implemented in Listing 11.3. The implementation has a poor runtime constant, but on large problems, the Karatsuba method will become substantially faster than the naive approach.

Listing 11.3: An arbitrary-precision binary integer class. Naive multiplication is implemented in an \( \Omega(n^2) \) manner, while fast Karatsuba multiplication is implemented in \( \Theta(n^{1.585...}) \).

```python
import numpy

def carry(array):
    n = len(array)
    for i in xrange(n-1):
        # we add numbers right to left:
```

import numpy

def carry(array):
    n = len(array)
    for i in xrange(n-1):
        # we add numbers right to left:
j=n-i-1

if array[j] > 1:
    array[j-1] += array[j] / 2
    array[j] = array[j] % 2

class BigInt:
    def __init__(self, bitstring):
        self.bitstring = list(bitstring)

        for x in self.bitstring:
            assert(x in (0,1))

        if len(self.bitstring) == 0:
            self.bitstring = [0]

    def trim_unneeded_bits(self):
        # remove leading [0,0,0,...] bits
        # e.g., [0,0,1,0,1] --> [1,0,1]
        n = len(self.bitstring)
        for i in xrange(n):
            if self.bitstring[i] == 1:
                break
        # do not trim if the value is [0].
        if self.bitstring != [0]:
            self.bitstring = self.bitstring[i:]

    def __add__(self, rhs):
        n=max(len(self.bitstring), len(rhs.bitstring))
        result=[0]*(1+n)

        for i in xrange(n):
            if i < len(self.bitstring):
                result[len(result)-i-1] += self.bitstring[len(self.bitstring)-i-1]
            if i < len(rhs.bitstring):
                result[len(result)-i-1] += rhs.bitstring[len(rhs.bitstring)-i-1]

        carry(result)
        res_big_int = BigInt(result)
        res_big_int.trim_unneeded_bits()
        return res_big_int

    def __sub__(self, rhs):
11.2. FAST MULTIPLICATION OF LONG INTEGERS

```python
rhs_comp = rhs.twos_complement(len(self.bitstring))
result = self + rhs_comp
result.bitstring[0] = 0
result.trim_unneeded_bits()

return result

def shifted_left(self, num_bits):
    return BigInt(self.bitstring + ([0]*num_bits))

def twos_complement(self, num_bits):
    padded = ([0]*(num_bits - len(self.bitstring))) + self.bitstring
    inverted = BigInt(padded)

    # flip bits:
    for i in xrange(len(inverted.bitstring)):
        if inverted.bitstring[i] == 0:
            inverted.bitstring[i] = 1
        else:
            inverted.bitstring[i] = 0

    # add 1:
    result = inverted + BigInt([1])
    return result

def __str__(self):
    bin_string = ''.join([str(b) for b in self.bitstring])
    return bin_string + ' = ' + str(int(bin_string, 2))

    # note: this cheats by letting each bit be >2 (they essentially use
    # python's native arbitrary precision integers). it is still \in
    # \Theta(n^2), but will have a decent runtime constant. it exists
    # for testing only.

def naive_mult(lhs, rhs):
    result = [0]*(len(lhs.bitstring)+len(rhs.bitstring))
    for i in xrange(len(lhs.bitstring)):
        for j in xrange(len(rhs.bitstring)):
            result[1+i+j] += lhs.bitstring[i]*rhs.bitstring[j]

    carry(result)
    result = BigInt(result)
    result.trim_unneeded_bits()
    return result
```
```python
# r(n) = 4 r(n/2) + \Theta(n)
# --> r(n) \in \Theta(n^{\log_2(4)}) = \Theta(n^2)
def recursive_mult(lhs, rhs):
    n = max(len(lhs.bitstring), len(rhs.bitstring))
    if len(lhs.bitstring) < len(rhs.bitstring):
        lhs = BigInt([0]*(n - len(lhs.bitstring))) + lhs.bitstring
    if len(rhs.bitstring) < len(lhs.bitstring):
        rhs = BigInt([0]*(n - len(rhs.bitstring))) + rhs.bitstring

    if n <= 2:
        return naive_mult(lhs, rhs)

    x_high = BigInt(lhs.bitstring[:n/2])
    x_low = BigInt(lhs.bitstring[n/2:])

    y_high = BigInt(rhs.bitstring[:n/2])
    y_low = BigInt(rhs.bitstring[n/2:])

    result = BigInt([0])
    z_low = recursive_mult(x_low, y_low)
    z_high = recursive_mult(x_high, y_high)
    t1 = recursive_mult(x_low, y_high)
    t2 = recursive_mult(x_high, y_low)
    z_mid = t1 + t2

    result += z_low
    msbs = n/2
    lsbs = n-n/2
    result += z_mid.shifted_left(lsbs)
    result += z_high.shifted_left(2*lsbs)

    return result

# r(n) = 3 r(n/2) + \Theta(n)
# --> r(n) \in \Theta(n^{\log_2(3)}) = \Theta(n^{1.585...})
def karatsuba_mult(lhs, rhs):
    n = max(len(lhs.bitstring), len(rhs.bitstring))
    if len(lhs.bitstring) < len(rhs.bitstring):
        lhs = BigInt([0]*(n - len(lhs.bitstring))) + lhs.bitstring
    if len(rhs.bitstring) < len(lhs.bitstring):
        rhs = BigInt([0]*(n - len(rhs.bitstring))) + rhs.bitstring

    if n <= 2:
        return naive_mult(lhs, rhs)
```
11.2. FAST MULTIPLICATION OF LONG INTEGERS

```python
x_high = BigInt(lhs.bitstring[:n//2])
x_low = BigInt(lhs.bitstring[n//2:])

y_high = BigInt(rhs.bitstring[:n//2])
y_low = BigInt(rhs.bitstring[n//2:])

result = BigInt([0])

z_low = karatsuba_mult(x_low, y_low)
z_high = karatsuba_mult(x_high, y_high)
z_mid = karatsuba_mult(x_low + x_high, y_low + y_high) - z_low - z_high

result += z_low
msbs = n//2
lsbs = n-n//2
result += z_mid.shifted_left(lsbs)
result += z_high.shifted_left(2*lsbs)

return result
```

```python
n=64
numpy.random.seed(0)
x=BigInt(numpy.random.randint(0,2,n))
y=BigInt(numpy.random.randint(0,2,n))
#x=BigInt([1,1,1])
#y=BigInt([1,0,0])
print 'x', x
print 'y', y
print ''

print 'recursive prod', recursive_mult(x, y)
print 'karatsuba prod', karatsuba_mult(x, y)
```

Even when implementing circuits that will multiply constant-precision integers (e.g., creating the hardware on the CPU that will multiply two 32-bit unsigned int types from C/C++), algorithms like Karatsuba’s method are important for creating circuits that implement practically fast multiplication between constant-precision integers without using a lot of silicon.

---

6Building a chip that simultaneously computes all $n^2$ digit products would use more silicon (which would cost substantially more money and be less power efficient) and may still not be as fast as a circuit using a divide-and-conquer strategy reminiscent of Karatsuba’s method.