

## Faster Multiplication

We have analysed the algorithms for addition and multiplication of multi-digit numbers as we learnt to do in school in term of the basic operations `add_mac` and `mul_mac` that do arithmetic with digits. We note that the multiplication algorithm takes time which is quadratic in the size of the numbers. Can we do it faster?

In 1960 Anatoly Karatsuba discovered an algorithm for multiplication that *improves* on this quadratic growth. Later this was improved further. The Toom-Cook algorithm is faster and the Schönhage-Strassen algorithm is faster than that. In 2019, the galactic algorithm was discovered by Harvey and van der Hoeven which grows like  $C \cdot n \log(n)$  where  $n$  is the size of the inputs. However, each of these algorithms is more complicated than the previous one. Perhaps as a consequence the constant  $C$  is very large. Hence, this algorithm currently remains of theoretical use only!

It is perhaps worth noting that faster algorithms are often more complicated and use more sophisticated mathematics than the “classical” ones!

### The idea

The key idea behind the Karatsuba algorithm is as follows. We have an identity

$$(a + b \cdot N) \cdot (c + d \cdot N) = a \cdot c + (a \cdot d + b \cdot c)N + b \cdot d \cdot N^2$$

This *appears* to need the calculation of 4 products  $a \cdot c$ ,  $a \cdot d$ ,  $b \cdot c$  and  $b \cdot d$ . However, we have the identity

$$a \cdot d + b \cdot c = (a + b) \cdot (c + d) - a \cdot c - b \cdot d$$

Thus, in order to calculate the product above, we only need to calculate the products  $a \cdot c$ ,  $b \cdot d$  and  $(a + b) \cdot (c + d)$ . *Assuming* (as is often the case) that addition and subtraction are not expensive operations, we have replaced 4 steps by 3 steps.

This alone, does not seem much of a saving. However, we can repeat this approach while calculating the products that we need to calculate by taking  $N = B^{2^{n-1}}$  for various values of  $n$ . As usual, for  $n = 1$ , we will use built-in operations for base  $B$ . Secondly, we pad  $\underline{a}$  and  $\underline{b}$  to be of length  $2^n$  to simplify our algorithm.

```
define kmul( $\underline{a}, \underline{b}$ ):  
     $k$  is the maximum of the lengths of  $\underline{a}$  and  $\underline{b}$   
     $n$  is the smallest integer such that  $2^n \geq k$   
    pad  $\underline{a}$  and  $\underline{b}$  by 0's on the right so that their length is  $2^n$   
    return kmul1( $\underline{a}, \underline{b}$ )
```

This uses the helper routine `kmul1` that works with multi-digit numbers of length  $l$  a power of 2 and generates output of length  $2l$ .

```

define kmul1(a,b):
    l = length of a (and b)
    if l ≤ 1:
        return mul_mac(a0,b0) as tuple of length 2
    c first l/2 elements of a as a tuple
    d remaining l/2 elements of a as a tuple
    e first l/2 elements of b as a tuple
    f remaining l/2 elements of b as a tuple
    g = add(c,d)
    h = add(e,f)
    r = kmul1(c,e)
    s = kmul1(d,f)
    t = kmul(g,h)
    u = sub(t,r)
    v = sub(u,s)
    s' obtained by shifting s to the right by l padded by 0's
    v' obtained by shifting v to the right by l/2 padded by 0's
    w = add(r,s')
    x = add(w,v')
    x' obtained by shifting x to the right padded by 0's to make it of length 2l
    return x

```

### Correctness

Since the algorithm is somewhat long, let us first verify its correctness. It is clear that **kmul** works correctly if **kmul1** works correctly.

Note that **kmul1** is to be called with arguments a and b of the same length *l* which *must* be a power of 2. Its output *must* then be of length *2l*.

We first take care of the case where this is  $l = 2^0$ . In that case, the algorithm just returns the result of the internal multiplication operation **mul\_mac** as a tuple of length 2.

When  $l > 1$ , we see that a and b are divided into halves of equal length  $l/2$ .

Thus, we have the equations:

$$\begin{aligned}\underline{a} &= \underline{c} + \underline{d} \cdot B^{2^{n-1}} \\ \underline{b} &= \underline{d} + \underline{e} \cdot B^{2^{n-1}}\end{aligned}$$

Assuming that `kmul1` works correctly for tuples of length  $l/2$ , we see that

$$\begin{aligned}\underline{r} &= \underline{c} \cdot \underline{d} \\ \underline{s} &= \underline{d} \cdot \underline{f}\end{aligned}$$

There is a small catch with  $\underline{t}$ ! Note that  $\underline{g}$  and  $\underline{h}$  could be of length  $1 + l/2$  due to carry-over in addition. This is not a power of 2 unless  $l = 2$ , so we need to use `kmul` to carry out the multiplication, since `kmul` carries out the padding if it is required. Assuming that this works correctly, we note that

$$\begin{aligned}\underline{g} &= \underline{c} + \underline{e} \\ \underline{h} &= \underline{e} + \underline{f} \\ \underline{t} &= \underline{g} \cdot \underline{h} = (\underline{c} + \underline{d}) \cdot (\underline{e} + \underline{f})\end{aligned}$$

At this point we make use of a routine `sub` that performs multi-precision subtraction. Assuming that it works correctly, we see that

$$\underline{v} = \underline{t} - \underline{r} - \underline{s} = \underline{c} \cdot \underline{f} + \underline{d} \cdot \underline{e}$$

by some algebra. Note that this is non-negative, so we need not worry about `sub` returning a negative number.

The right shifts used to obtain  $\underline{s}'$  and  $\underline{v}'$  correspond to multiplication by  $B^{2^n}$  and  $B^{2^{n-1}}$  respectively. This (along with the correctness of `add!`) ensures that we have the identity

$$\underline{x} = \underline{r} + \underline{v} \cdot B^{2^{n-1}} + \underline{s} \cdot B^{2^n}$$

Using the above equations, we see that this means that  $\underline{x}'$  is the desired answer since we also ensure that its length is  $2l$  as required. By the above identity and the usual inequalities we note that its length cannot be greater than  $2l = 2^{n+1}$ .

We note that to make this algorithm work properly we need `add` and `sub` to return answers of the shortest possible length by removing 0's on the right of their answers.

### Time Complexity

In the following we measure time complexity by *only* counting the number of times `mul_mac` is used. In other words, we *assume* that `add`, `sub` and book-keeping operations do not significantly add to the time. This assumption is reasonable for many computing devices. However, one can also note that the time added is *linear* in the length of the input and thus, as we shall see later, does not change the *dominant* term of the answer.

Suppose that  $M(p, q)$  is the amount of time taken by `kmul` for inputs of size  $p$  and  $q$ . We let  $n$  to be the smallest integer such that  $2^n \geq \max\{p, q\}$ . From the above description  $M(p, q) \leq N(l)$  where  $N(l)$  denotes the amount of time taken by `kmul1` on input of size  $l = 2^n$ .

Note that  $N(1) = 1$  so we need to calculate  $N(l)$  for  $l = 2^n$  where  $n \geq 1$ .

For the moment, we ignore the fact that  $\underline{t} = \text{kmul}(\underline{g}, \underline{h})$  multiplies numbers that could be of length  $1 + l/2$ . Instead we assume that this is just a call to `kmul1`, for simplicity. In that case, we get  $N(l) = 3 \cdot N(l/2)$ .

It follows easily that  $N(l) = 3^n$  when  $l = 2^n$ . Put differently,

$$N(l) = 3^{\log_2(l)} = 2^{\log_2(3) \cdot \log_2(l)} = l^{\log_2(3)}$$

Since  $\log_2(3) < \log_2(4) = 2$ , we see this is a *smaller* power of  $l$  than quadratic.

In order to complete the analysis, we must justify the replacement of `kmul` by `kmul1` in the above analysis. We also resolve this by modifying the algorithm by introducing some additional book-keeping.

### Fixing the algorithm.

One way to fix the above algorithm so as to only use multiplication of length  $l/2$  tuples as a step in multiplication of length  $l$  is as follows.

We have the identity

$$a \cdot d + b \cdot c = (a - b) \cdot (d - c) + a \cdot c + b \cdot d$$

Note that  $|a - b|$  and  $|d - c|$  are both between 0 and  $B^{l/2} - 1$ . Moreover, if  $s$  denotes the *sign* of  $(a - b)$  and  $t$  denotes the *sign* of  $d - c$ , then we have

$$a \cdot d + b \cdot c = (a \cdot c + b \cdot d) + (s \cdot t)|a - b| \cdot |d - c|$$

All 3 of the products of multi-digit numbers on the right can be computed using `kmul1` for tuples of length  $l/2$  provided  $a, b, c$  and  $d$  have length  $l/2$ .

```
define kmul1(a, b):
    l = length of a (and b)
    if l ≤ 1:
        return mul_mac(a0, b0) as tuple of length 2 or 1
```

c first  $l/2$  elements of a as a tuple  
d remaining  $l/2$  elements of a as a tuple  
e first  $l/2$  elements of b as a tuple  
f remaining  $l/2$  elements of b as a tuple

$$(g, sign_g) = \text{sub}(c, d)$$

$$(\underline{h}, \text{sign}_h) = \text{sub}(\underline{f}, \underline{e})$$

$$\text{sign} = \text{sign}_g \cdot \text{sign}_h$$

$$\begin{aligned}\underline{r} &= \text{kmul1}(\underline{c}, \underline{e}) \\ \underline{s} &= \text{kmul1}(\underline{d}, \underline{f}) \\ \underline{t} &= \text{kmul1}(\underline{g}, \underline{h})\end{aligned}$$

$$\begin{aligned}\underline{u} &= \text{add}(\underline{r}, \underline{s}) \\ \text{if } \text{sign} &= +1: \\ &\quad \underline{v} = \text{add}(\underline{u}, \underline{t}) \\ \text{else:} \\ &\quad (\underline{v}, \text{sign}_v) = \text{sub}(\underline{u}, \underline{t})\end{aligned}$$

$\underline{s}'$  obtained by shifting  $\underline{s}$  to the right by  $l$  padded by 0's  
 $\underline{v}'$  obtained by shifting  $\underline{v}$  to the right by  $l/2$  padded by 0's

$$\begin{aligned}\underline{w} &= \text{add}(\underline{r}, \underline{s}') \\ \underline{x} &= \text{add}(\underline{w}, \underline{v}')\end{aligned}$$

$\underline{x}'$  obtained by shifting  $\underline{x}$  to the right padded by 0's to make it of length  $2l$

**return**  $\underline{x}$

To complete this description, we need an algorithm **sub** that takes two multi-digit numbers  $\underline{a}$  and  $\underline{b}$  returns  $|\underline{a} - \underline{b}|$  and the sign of  $\underline{a} - \underline{b}$ .

### Subtraction

In order to implement subtraction we can work with the notion of  $(B - 1)$ -complements as follows. Note that if  $\underline{a} = (a_0, \dots, a_n)$  and if we define  $\underline{z} = (z_0, \dots, z_n)$  by  $a_i + z_i = (B - 1)$ , we have the identity

$$B^{n+1} - 1 = \underline{a} + \underline{z}$$

Now, given any multi-digit number  $\underline{b}$  of length  $n + 1$ , we note

$$B^{n+1} - 1 = (\underline{a} - \underline{b}) + (\underline{z} + \underline{b})$$

If  $\underline{r} = \underline{z} + \underline{b}$  and  $r_{n+1} = 0$ , then this shows that  $\underline{a} - \underline{b}$  is the  $(B - 1)$ -complement of  $\underline{z} + \underline{b}$ . On the other hand, if  $r_{n+1} = 1$ , then  $\underline{r} = B^{n+1} + \underline{r}'$  where  $\underline{r}' = (r_0, \dots, r_n)$ . In this case, we have the equation

$$\underline{b} - \underline{a} = \underline{r}' + 1$$

so that we have  $\underline{a} - \underline{b}$  is the negative of  $\underline{r}' + 1$ .

To use this we need an operation **com\_mac** implemented at machine level that gives the  $(B - 1)$ -complement of a digit. Using this we can implement **sub** by using addition of  $\underline{z}$  and  $\underline{b}$ .

```

define commac(a):
    return base − 1 − a

define sub(a, b):
    Find the maximum n of lengths of a and b.
    Pad a and b by 0's on the right to make them of length n.
    Calculate the B − 1 complement z of a.
    r = add(z, b)
    if rn+1 = 0:
        Calculate the B − 1 complement s of r.
        return s and indicate that the sign is +1.
    else:
        r' is (r0, ..., rn).
        Calculate s = r' + 1.
        return s and indicate that the sign is −1.

```

**Exercise:** Verify the correctness of this algorithm using the discussion above.  
 (In particular, note that there is no overflow when we add 1 to *r*.)

**Exercise:** Calculate the time complexity of this algorithm by counting the usage of add<sub>mac</sub>. (In particular, count the time complexity of adding 1 to *r*. Can this be reduced by using an algorithm different from add?)