Analysis of algorithms

We have written 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.

Addition. This algorithm was represented by the following pseudo-code:

```
define add1($\underline{a}, \underline{b}, c$):
    $\underline{r}$ = add_mac($a_0, b_0$)
    $\underline{s}$ = add_mac($r_0, c$)
    $\underline{t}$ = add_mac($r_1, s_1$)
    if $\underline{a}$ and $\underline{b}$ are of size 1:
        return ($s_0, t_0$)
    return list with first entry $s_0$ and remaining entries from add1($\underline{a'}, \underline{b'}, t_0$)

define add($\underline{a}, \underline{b}$):
    Find the maximum $m$ of lengths of $\underline{a}$ and $\underline{b}$.
    Pad $\underline{a}$ and $\underline{b}$ by $0$'s on the right to make them of length $m$.
    return add1($\underline{a}, \underline{b}, 0$)
```

Multiplication. This algorithm was represented by the following pseudo-code:

```
define mul1($a, \underline{b}, c$):
    $\underline{r}$ = mul_mac($a, \underline{b}_0$)
    $\underline{s}$ = add_mac($r_0, c$)
    $\underline{t}$ = add_mac($r_1, s_1$)
    if $\underline{b}$ is of size 1:
        return ($s_0, t_0$)
    return list with first entry $s_0$ and remaining entries from mul1($a, \underline{b'}, t_0$)

define mul($\underline{a}, \underline{b}$):
    $\underline{r}$ = mul1($a_0, \underline{b}, 0$)
    if $\underline{a}$ is of size 1:
        return $\underline{r}$
    $\underline{s'}$ = mul($\underline{a'}, \underline{b}$)
    Extend $\underline{s'}$ by a $0$ on the left to get $\underline{s}$.
    return add($\underline{r}, \underline{s}, 0$)
```

Our next task is to *analyse* these algorithms with two goals in mind:

Correctness: We must ensure that these algorithms give the *mathematically correct* answer in all cases.

Efficiency: We must count the amount of time (and possibly space) that these algorithms use.

Proof of Correctness

It is reasonably clear that `add` works correctly if `add1` complies with the following specification:

> `add1`$(\underline{a}, \underline{b}, c)$ adds the multi-digit numbers $\underline{a}$ and $\underline{b}$ of the *same* length with the single digit number c.

As before, if $\underline{a} = (a_0, \ldots, a_n)$ and $\underline{b} = (a_0, \ldots, a_n)$, then as before we define $\underline{a'} = (a_1, \ldots, a_n)$ and $\underline{b'} = (b_1, \ldots, b_n)$. We then verify the following identities.

$$\begin{aligned}
\underline{a} + \underline{b} + c = (a_0 + b_0 + c) + (\underline{a'} + \underline{b'}) \cdot B \\
= (r_0 + r_1 \cdot B + c) + (\underline{a'} + \underline{b'}) \cdot B \\
= (s_0 + s_1 \cdot B + r_1 \cdot B) + (\underline{a'} + \underline{b'}) \cdot B \\
= s_0 + (t_0 + t_1 \cdot B) \cdot B + (\underline{a'} + \underline{b'}) \cdot B \\
= s_0 + (t_1 \cdot B) \cdot B + (\underline{a'} + \underline{b'} + t_0) \cdot B
\end{aligned}$$

Now $s_0 + (\underline{a'} + \underline{b'} + t_0) \cdot B$ is what the `add1` program reduces the calculation to. Thus, the given method for calculation works correctly by induction on n *provided* $t_1 = 0$ for *all* possible choices of a_0, b_0 and c in the range $[0, B-1]$. Clearly, $t_1 = 0$ if $a_0 + b_0 + c_0 < B^2$. Now, the maximum value of $a_0 + b_0 + c_0$ is $3(B-1)$. So we need the inequality $3(B-1) < B^2$ which we check holds for all $B \geq 2$.

Thus, we have verified the correctness of `add1` and `add`.

With notation as above, the following formula holds.

$$\underline{a} \cdot \underline{b} == a_0 \cdot \underline{b} + (\underline{a'} \cdot \underline{b}) \cdot B$$

The `mul` algorithm calculates the first term using `mul1` as $\underline{r}$ and the parenthesised part of the second term using `mul` inductively as $\underline{s'}$. The two terms are then added using `add`. It then follows by induction on the length of $\underline{a}$ that `mul` works correctly if `add` works correctly and `mul1` complies with the following specification:

> `mul1`$(a, \underline{b}, c)$ multiplies the multi-digit number $\underline{b}$ by the digit a and adds the single digit number c to the product.

We now verify the following identities.

$$\begin{aligned}
a \cdot \underline{b} + c = (a \cdot b_0 + c) + (a \cdot \underline{b'}) \cdot B \\
= (r_0 + r_1 \cdot B + c) + (a \cdot \underline{b'}) \cdot B \\
= (s_0 + s_1 \cdot B + r_1 \cdot B) + (a \cdot \underline{b'}) \cdot B \\
= s_0 + (t_0 + t_1 \cdot B) \cdot B + (a \cdot \underline{b'}) \cdot B \\
= s_0 + (t_1 \cdot B) \cdot B + (a \cdot \underline{b'} + t_0) \cdot B
\end{aligned}$$

Now $s_0 + (a \cdot \underline{b'} + t_0) \cdot B$ is what the `mul1` program reduces the calculation to. Thus, the given method for calculation works correctly by induction on n

provided $t_1 = 0$ for *all* possible choices of a_0, b_0 and c in the range $[0, B-1]$. Clearly, $t_1 = 0$ if $a_0 \cdot b_0 + c_0 < B^2$. Now, the maximum value of $a_0 \cdot b_0 + c_0$ is $(B-1)^2 + (B-1) = B(B-1)$. So we need the inequality $B(B-1) < B^2$ which we check holds for all $B \geq 2$.

Thus, we have verified the correctness of `mul1` and `mul`.

Calculating the complexity

We make the simplifying assumption that we *only* need to calculate the calls to `add_mac` and `mul_mac`. In other words, the remaining tasks of labelling data and moving it around are treated as "instantaneous".

Addition algorithm. Looking at `add`, we see that it results in a call to `add1` where $\underline{a}$ and $\underline{b}$ are padded by 0's to size m which is the maximum of their sizes. If $m = 1$, then `add1` returns the answer after exactly 3 calls to `add_mac`. On the other hand, if $m > 1$, then after these three calls we again call `add1` with numbers of size $m - 1$. Thus, if $A(m)$ denotes the number of `add_mac` calls in order to complete the execution of `add1` on numbers of size m, we have

$$A(1) = 3 \text{ and } A(m) = 3 + A(m-1) \text{ for} m > 1$$

We easily see that this means that $A(m) = 3m$.

Note that m is the maximum of the size of the inputs, Note also that this is roughly the logarithm to the base B of the *integers* that $\underline{a}$ and $\underline{b}$ represent. This is a common aspect of the study of complexity of semi-numerical algorithms, which is that we study this as a function of the logarithm of the integers that are represented.

Secondly, we note that there may be cases where `add1` is called with $c = 0$. In that case, we can avoid the second and third calls to `add_mac`. This can be used to "squeeze" the maximum optimality of the algorithm if really required. However, this will only reduce the 3 in equation $A(m) = 3m$ to a smaller constant. For us the important thing is:

> The time complexity of the addition algorithm grows *linearly* as a function of the logarithm of the numbers being added.

Multiplication algorithm. We note that the multiplication algorithm makes use of `add_mac` *and* `mul_mac`. So we need to count the calls to each of them. Secondly, we are *not* padding $\underline{a}$ or $\underline{b}$, so we need to write functions that depend on each of their lengths' say these are p and q respectively. Thus, we wish to compute two functions:

$M_a(p, q)$**:** The number of calls to `add_mac` while carrying out the algorithm `mul` on inputs of size p and q.

$M_m(p, q)$**:** The number of calls to `mul_mac` while carrying out the algorithm `mul` on inputs of size p and q.

These will depend on corresponding functions for `mul1`. (Note that the only input of variable length to `mul1` is $\underline{b}$.)

$N_a(q)$**:** The number of calls to `add_mac` while carrying out the algorithm `mul1` when input $\underline{b}$ is of size q.

$N_m(p,q)$**:** The number of calls to `mul_mac` while carrying out the algorithm `mul1` when input $\underline{b}$ is of size q.

Looking at `mul`, we see that it results in a call to `mul1` if $p = 1$. Thus, to begin with we see that $M_a(1,q) = N_a(q)$ and $M_m(1,q) = N_m(q)$. When $p > 1$, we will, in addition call `mul` with input of size $p-1$ and q and also call `add` at the end. What is the size of the inputs to `add`? We note that the product of a p digit number and a q digit number is at most $p+q$ digits. Thus, $\underline{r}$ is of length $1+p$ and $\underline{s}$ is of length $1+(p-1)+q = p+q$. The maximum of these is $p+q$. Thus, we get the following equations for $p > 1$:

$$\begin{aligned} M_a(p,q) &= N_a(q) + M_a(p-1,q) + A(p+q) \\ M_m(p,q) &= N_m(q) + M_m(p-1,q) \end{aligned}$$

Looking at `mul1` we see that for $q = 1$ it has one call to `mul_mac` and 2 calls to `add_mac`. When $q > 1$, it also recursively calls itself with input of size $q-1$. Thus, we get the following equations:

$$\begin{aligned} N_m(1) &= 1 \\ N_a(1) &= 2 \\ N_m(q) &= 1 + N_m(q-1) \text{ for } q > 1 \\ N_a(q) &= 2 + N_a(q-1) \text{ for } q > 1 \end{aligned}$$

Using these equations, we see that $N_m(q) = q$ and $N_a(q) = 2q$. Plugging these values and the value for $A(p+q)$ in the previous equations, we get

$$\begin{aligned} M_m(1,q) &= q \\ M_a(1,q) &= 2q \\ M_m(p,q) &= q + M_m(p-1,q) \text{ for } p > 1 \\ M_a(p,q) &= 2q + M_a(p-1,q) + 3(p+q) \text{ for } p > 1 \end{aligned}$$

Using these equations, we see that

$$\begin{aligned} M_m(p,q) &= p \cdot q \\ M_a(p,q) &= 2p \cdot q + 3(p^2 + p - 2)/2 + 3(p-1)q \end{aligned}$$

(Note that this uses the formula $p + (p-1) + \cdots + 2 = (p^2 + p + 2)/2$.) The key point to note is that the growth is *quadratic*.