Algorithms for addition and multiplication

Let's try to remember your first experience with numbers, way back when you were a child in grade school. In grade 1, you learned how to count up to ten and to do basic arithmetic using your fingers. When doing so, you memorized sums of *single digit* numbers ($4 + 7 = 11$, $3 + 6 = 9$, etc). Later on in about grade 4, you learned a method for adding *multiple digit* numbers, which was based on the single digit additions that you had memorized. For example, you were asked to compute things like:

$$\begin{array}{rr} & 2343 \\ + & \underline{4519} \\ & ? \end{array}$$

The method that you learned was a sequence of computational steps, commonly called an *algorithm*. What was the algorithm ? Let's call the two numbers a and b and let's say they have N digits each. Then the two numbers can be represented as an array of single digit numbers $a[\,]$ and $b[\,]$. We can define a variable $carry$ and compute the result in an array $r[\,]$. You know how this works. You go column by column, adding the pair of single digit numbers in that column and adding the carry (0 or 1) from the previous column. We can write the algorithm in *pseudocode*[1] as follows:

Algorithm 1 Addition (base 10): Add two N digit numbers a and b which are represented as arrays of digits

```
carry = 0
for i = 0 to N − 1 do
    r[i] ← (a[i] + b[i] + carry) % 10
    carry ← (a[i] + b[i] + carry)/10
end for
r[N] ← carry
```

The operator % is the "mod" operator. It computes the remainder of the division. The operator / ignores the remainder, i.e. it rounds down (often called the "floor").

Also note that the above algorithm requires that you can compute (or look up in a table that you have "memorized") the sum of two single digit numbers with '+' operator, and also (possibly) add 1 to that result.

Later on in grade school, you learned how to multiply two numbers. Again, you first memorized a multiplication table for single digit numbers (e.g. $6 \times 7 = 42$). You then learned a sequence of steps for multiplying a pair of N digit numbers. The algorithm is shown on the next page.

There are two stages to the algorithm. The first is to compute a 2D array whose rows contain the first number a multiplied by the single digits of the second number b (times the corresponding power of 10). The second stage is to add up the rows of this 2D array.

Note: in the example below, I have not shown various "carries" that were used to compute the 2D array and the final result.

[1] not code in a real programming language, but good enough for communicating between humans i.e. me and you

```
   352
 x 964
-------
  1408
 21120
316800
-------
339328
```

Algorithm 2 Multiplication (base 10) of two numbers a and b

for $j = 0$ to $N - 1$ **do**
  $carry \leftarrow 0$
  for $i = 0$ to $N - 1$ **do**
    $prod \leftarrow (a[i] * b[j] + carry)$
    $tmp[j][i + j] \leftarrow prod\%10$
    $carry \leftarrow prod/10$
  end for
  $tmp[j][N + j] \leftarrow carry$
end for
for $i = 0$ to $2 * N - 1$ **do**
  $sum \leftarrow carry$
  for $j = 0$ to $N - 1$ **do**
    $sum \leftarrow sum + tmp[j][i]$
  end for
  $r[i] \leftarrow sum\%10$
  $carry \leftarrow sum/10$
end for
$r[2 * N] \leftarrow carry$

Analysis of Algorithm

Let's compare the addition and multiplication algorithms in terms of the number of operations required. The addition algorithm involves a single **for** loop which is run N times. For each pass through the loop, there is a fixed number of simple operations. There are also a few operations that are performed outside the loop. We would say that the addition algorithm requires $c_1 + c_2 N$ operations, i.e. a constant c_1 plus a term that is proportional (with factor c_2) to the number N of digits. We are ignoring the details that define c_1 and c_2 which have to do with the actual machine implementation of the various instructions. (You will learn about this in COMP 273.)

We saw that the multiplication algorithm involves two components, each having a pair of **for** loops, one inside the other. This "nesting" of loops leads to N^2 passes through the operations within the inner loop. For each pass, there are various basic operations performed.

Suppose we consider the first component, in which we produce the two dimensional matrix *tmp*. Suppose some number (say c_3) of operations are inside both **for** loops, some number (say c_4) of

operations are inside just one of the **for** loops, and some number (say c_5) of operations are outside both **for** loops. Then the number of operations is $c_5 + c_4N + c_3N^2$. The same argument would apply for the second step of the multiplication algorithm, since again we have two nested **for** loops.

To summarize, the number of operations taken by the addition and multiplication algorithms depends both on the c values as well as on the number of digits N. *It is very important to realize that, for large N, multiplication requires far more operations than additions since N^2 will dominate over N, regardless of the particular values of the c's.*

Let's next consider the space that is required to perform the above algorithms. The addition algorithm used arrays $a[], b[], r[]$ which are each of size N so the space required grows with N. The multiplication algorithm used a 2D array $tmp[][]$ which grows with N^2. When N is very large, the space requirements would be significant. Can this be avoided? Yes, it can.

Rather than making a 2D table and then summing up the rows after the table is constructed, you could build the rows one at a time and use the $r[]$ vector to acculumate the sum *as you build the rows*. Once a row has been added to the sum, there is no reason to keep it around. Notice that this does not speed up the algorithms. The total number of operations still grows with N^2, since you ultimately are adding up N rows which each have N digits. But you are saving significantly on space.

Subtraction

In grade school, you also learned how to perform subtraction. Subtraction was more difficult to learn than addition since you needed to learn the trick of borrowing, which is the opposite of carrying. In the example below, you needed to write down the result of 2-5, but this is a negative number and so instead you change the 9 to an 8 in the first number and you change the 2 to a 12, then compute 12-5=7 and write down the 7.

```
   924
 -  352
 -------
   572
```

This "borrowing" trick is straightforward (easy to say now!). In Assignment 1, you will be asked to code up an algorithm for doing this.

Long Division

The last basic arithmetic operation you learned in grade school was long division. Suppose you want to compute `41672542996 / 723`. Again you learned a method for doing this.

```
        5 ...
    --------------
723 |  41672542996
       3615
       ----
        552  ...
```

In the example above, you asked yourself, does 723 divide into 416? No. So, then you figure out how many times 723 divides into 4167. The answer is 5. You multiply 723 by 5 and then subtract this from 4167, etc, etc.

Why does this work? How would you write it down as an algorithm? (In Assignment 1, you will be required to code it up.)

Arithmetic in other bases?

You are used to representing positive integers in base 10, that is, as a sum of powers of 10. But there is nothing special about the number 10. In principle, you can represent positive integers as the sum of numbers in any base.

Let's consider a few examples of representing numbers in base 8, that is, as a sum of powers of 8. So,

$$(\ a[N-1]\cdots a[2]\ a[1]\ a[0]\ )_8\ =\ a[N-1]*8^{N-1}+\cdots+a[2]*8^2+a[1]*8+a[0]$$

where the $a[i]$ are all in $\{\ 0,1,2,\cdots,7\}$. For example,

$$(736)_8 = 7*8^2+3*8+6 = (478)_{10}$$

and

$$(1205)_8 = 1*8^3+2*8^2+0*8+5 = (645)_{10}.$$

The addition and multiplication algorithms we saw earlier work similarly as before. See if you can carry out the following sum and product of two numbers which are written in base 8. (Note: For technical reasons having to do with a typesetting hassle, I have not put the subscript 8 below which indicates base 8. But you should consider it there!)

```
  (1205)
+ (736)
-------
  (2143)      which is 1123 in base 10.
```

```
   (1205)
 * (736)
 -------
   (7436)
  (3617 )
 (10643  )
-----------
 (1132126)    which is 308310 in base 10.
```