## Addition

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 and you learned how to add single digit numbers. (4 + 7 = 11, 3 + 6 = 9, etc). Soon after, you learned a method for adding *multiple digit* numbers, which was based on the single digit additions that you had memorized. For example, you learned to compute things like:

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

The method that you learned was a sequence of computational steps: 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 $r[\ ]$.

```
           a[N-1]  .. a[0]
         + b[N-1]  .. b[0]
          ---------------
      r[N] r[N-1] ..  r[0]
```

The algorithm goes 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]

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

```
$carry = 0$
for $i = 0$ to $N-1$ do
    $r[i] \leftarrow (a[i] + b[i] + carry)\ \%\ 10$
    $carry \leftarrow (a[i] + b[i] + carry)\ /\ 10$
end for
$r[N] \leftarrow carry$
```

The operator / is integer division, and ignores the remainder, i.e. it rounds down (often called the "floor"). The operator % denotes the "mod" operator, which is the remainder of the integer division. (By the way, note the order of the remainder and carry assignments! Switching the order would be incorrect – think why.)

Also note that the above algorithm requires that you can compute (or look up in a table or memorized) the sum of two single digit numbers with '+' operator, and also add a carry of 1 to that result. You learned those single digit sums when you were very little, and you did so by counting on your fingers.

[1] By "pseudocode", I mean something like a computer program, but less formal. Pseudocode is not written in a real programming language, but good enough for communicating between humans i.e. me and you.

## Subtraction

Soon after you learned how to perform addition, you 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 and you change the 2 to a 12, then compute 12-5=7 and write down the 7.

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

The borrowing trick is similar to the carry trick in the addition algorithm. The borrowing trick allows you to perform subtraction on $N$ digit numbers, regardless on how big $N$ is. In Assignment 1, you will be asked to code up an algorithm for doing this.

## Multiplication

Later on in grade school, you learned how to multiply two numbers. For two positive integers $a$ and $b$,

$$a \times b = a + a + a + \cdots + a$$

where there are $b$ copies on the right side. In this case, we say that $a$ is the *multiplicand* and $b$ is the *multiplier*.

ASIDE: The above summation can also be thought of geometrically, namely consider a rectangular grid of tiles with $a$ rows and $b$ columns. You understood that the number of tiles is that same if you were to write it as $b$ rows of $a$ tiles. This gives you the intuition that $a \times b = b \times a$, a fact which is not at all obvious if you take only the summation above.

Anyhow, the summation definition above suggests an algorithm for computing $a \times b$: I claim this algorithm is very slow. To see why, think of how long it would take you $a$ and $b$ each had several digits. e.g. if $a = 1234$ and $b = 6789$, you would have to perform 6789 summations!

**Algorithm 2** Slow multiplication (by repeated addition).

```
product = 0
for i = 1 to b  do
  product ← product + a
end for
```

To perform multiplication more efficiently, one uses a more sophisticated algorithm, which you learned in grade school. An example of this sophisticated algorithm is shown here.

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

Notice that 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). This requires that you can compute single digit multiplication, e.g. 6 $\times$ 7 = 42. As a child, you learned a "lookup table" for this, usually called a "multiplication table". The second stage of the algorithm required adding up the rows of this 2D array.

**Algorithm 3** Grade school multiplication (using powers of 10)

```
for $j = 0$ to $N - 1$ do
  $carry \leftarrow 0$
  for $i = 0$ to $N - 1$ do
    $prod \leftarrow (a[i] * b[j] + carry)$
    $tmp[j][i + j] \leftarrow prod\%10$        // assume $tmp[][]$ was array initialized to 0.
    $carry \leftarrow prod/10$
  end for
  $tmp[j][N + j] \leftarrow carry$
end for
$carry \leftarrow 0$
for $i = 0$ to $2 * N - 1$ do
  $sum \leftarrow carry$
  for $j = 0$ to $N - 1$ do
    $sum \leftarrow sum + tmp[j][i]$        // could be more efficient here since many $tmp[][]$ are 0.
  end for
  $r[i] \leftarrow sum\%10$
  $carry \leftarrow sum/10$
end for
```

Of course, when you were a child, your teacher did not write out this algorithm for you. Rather, you saw examples, and you learned the pattern of what to do. Your teacher explained why this algorithm did what it was supposed to, and probably you got the main idea.

## Division

The fourth basic arithmetic operation you learned in grade school was division. Given two positive integers $a, b$ where $a > b$, the integer division $a/b$ can be thought of as the number of times (call it $q$) that $b$ can be subtracted from $a$ until the remainder is a positive number less than $b$. This gives

$$a = q\ b + r$$

where $0 \leq r < b$, where $q$ is called the *quotient* and $r$ is called the *remainder*. Note that if $a < b$ then quotient is 0 and the remainder is $a$. Also recall that $b$ is called the divisor and $a$ is called the *dividend* so

$$dividend = quotient * divisor + remainder.$$

The definition of $a/b$ as a repeated subtraction suggests the following algorithm:

**Algorithm 4** Slow division (by repeated subtraction):

```
$q = 0$
$r = a$
while $r \geq b$ do
    $r \leftarrow r - b$
    $q \leftarrow q + 1$
end while
```

This repeated subtraction method is very slow if the quotient is large. There is a faster algorithm which uses powers of 10, similar in flavour to what you learned for multiplication. This faster algorithm is of course called "long division".

### Example of long division

Suppose $a$ = `41672542996` and $b$ = `723`. Back in grade school, you learned how to efficiently compute $q$ and $r$ such that $a = qb + r$. The algorithm started off like this: (please excuse the dashes used to approximate horizontal lines)

```
             5

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

In this example, you asked yourself, does 723 divide into 416? The answer is No. So, then you figure out how many times 723 divides into 4167. You guess 5, by reasoning that $7 * 5 < 41$ whereas $7 * 6 > 41$. You multiply 723 by 5 and then subtract this from 4167, and you get a result between 0 and 723 so now you know that 5 was a good guess.

To continue with the algorithm beyond what I wrote above, you "bring down the 2" (that is, the underlined 2 in the dividend $4167\underline{2}542996$) and then you figure out how many times 723 goes into 5522, where the second 2 is the one you brought down.

*Why does this algorithm work?* Your teacher may have explained it to you, but my guess is that you didn't understand any better than I did. How would you write this algorithm down in pseudocode, similar to what I did with multiplication? In Assignment 1, you will not only have to write down pseudocode, you will be asked to code it up in Java.

## Computational Complexity of Grade School Algorithms

For the multiplication and division operations, I presented two versions each and I argued that one was fast and one was slow. We would like to be more precise about what we mean by this. In general we would like to discuss how long an algorithm takes to run. You might think that we want an answer to be in *seconds*. However, for such a real measure we would need to specify details about the programming language and the computer that we are using. We would like to avoid these real details, because languages and computers change over the years, and we would like our theory not to change with it.

The notion of speed or *complexity* of an algorithm in computer science is not measured in real time units such as seconds. Rather, it is measured as a mathematical function that depends on the *size of the problem*. In the case of arithmetic operations on two integers a, b which have $N$ digits each, we say that the size of the problem is $N$.

Let's briefly compare the addition and multiplication algorithms in terms of the number of operations required, in terms of the number of digits $N$. The addition algorithm has some instructions which are only executed once, and it has a `for` loop which is run $N$ times. For each pass through the loop, there is a fixed number of simple operations which include $\%, /, +$ and assignments $\leftarrow$ of values to variables. Let's say that the instructions that are executed just once take some constant time $c_1$ and let's say that the instructions that are executed within each pass of the `for` loop take some other constant $c_2$. We could try to convert these constants $c_1$ and $c_2$ to units of seconds (or nanoseconds!) if had a particular implemenetation of the algorithm in some programming language and we were running it on some particular computer. But we won't do that because this conversion is beside the main point. The main point rather is that these constants have *some* value which is independent of the variable $N$. To summarize, 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. A key concept which we will elaborate in a few weeks is that, for large values of $N$, the dominating term will be the last term. We will say that the algorithm runs in time "order" of $N$, or $O(N)$.

What about the multiplication algorithm? We saw that the multiplication algorithm involves two main steps, each having a pair of `for` loops, one nested inside the other. This "nesting" leads to $N^2$ passes through the inner loop. There are some instructions that executed in the outer `for` loops but not in the inner `for` loops, and these instructions are executed $N$ times. There are also some instructions that are executed outside of all the `for` loops and these are executed a constant number of times. Let $c_3$ be the constant amount of time (say in nanoseconds) that it takes to execute all the instructions that are executed just once, and let $c_4$ be the constant amount of time that it takes to execute all the instructions that are executed $N$ times, and let $c_5$ be the constant amount of time that it takes to execute all the instructions that are executed $N^2$ times, we have that the total amount of time taken by the multiplication algorithm is

$$c_3 + c_4 N + c_5 N^2.$$

Unlike with the addition algorithm, for large values of $N$, now the dominating term will be the $N^2$ term. We will say that the algorithm runs in time $O(N^2)$. We will see a formal definition of $O(\ )$ in a few weeks, once we have seen more examples of different algorithms and spent more time discussing their complexity.