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:

$$\begin{array}{c}
2343 \\
+ \ 4519 \\
\hline
\end{array}$$

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 $r[]$.

$$\begin{array}{c}
a[N-1] \ldots a[0] \\
+ b[N-1] \ldots b[0] \\
\hline
r[N] \ r[N-1] \ldots \ r[0]
\end{array}$$

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.

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

```plaintext
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 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.

---

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.

\[
\begin{array}{c}
924 \\
- 352 \\
\hline
572
\end{array}
\]

This “borrowing” trick was more sophisticated than the counting method, but also very powerful. (The counting method was fine if you had very small numbers and you were counting with your fingers. But it didn’t generalize well to large numbers.) Similar to the carry trick in the addition algorithm 2, 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. You first learned the definition of multiplication, and this was done by showing you a 2D grid of tiles with \( a \) rows and \( b \) columns. You understood that the number of tiles in the grid was called \( a \times b \), and you could get that number by adding the tiles row-by-row, or column-by-column (and so \( a \times b = b \times a \)).

Algorithm 2 Slow multiplication (by repeated addition).

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

Note how slow it is if \( a \) and \( b \) are large numbers, say with \( N \) digits where \( N \) is large. I will have more to say about this later in the lecture.

To perform multiplication more efficiently, we use a more sophisticated algorithm. An example of this sophisticated algorithm is shown here. We learned this algorithm in grade school.

\[
\begin{array}{c}
352 \\
* 964 \\
\hline
1408 \\
21120 \\
316800 \\
\hline
339328
\end{array}
\]
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)

```plaintext
for $j = 0$ to $N - 1$ do
    carry ← 0
    for $i = 0$ to $N - 1$ do
        prod ← $(a[i] \times b[j] + carry)$
        $tmp[j][i + j] \leftarrow prod \% 10$ // assume $tmp[][]$ was array initialized to 0.
        carry ← prod/10
    end for
    $tmp[j][N + j] \leftarrow carry$
end for

carry ← 0
for $i = 0$ to $2 \times N - 1$ do
    sum ← carry
    for $j = 0$ to $N - 1$ do
        sum ← sum + $tmp[j][i]$ // could be more efficient here since many $tmp[][]$ are 0.
    end for
    $r[i] \leftarrow sum \% 10$
    carry ← sum/10
end for
```

[ASIDE: I originally posted one more line outside the 2nd for loop, namely $r[2N] \leftarrow carry$. However, one can show that this value of carry always is 0. To see this, note $(10^n - 1) \times (10^n - 1) = 10^{2n} - 2 \times 10^n + 1 < 10^{2n}$]

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 perhaps you understood.

**Division**

The fourth basic arithmetic operation you learned in grade school was division. Given two positive integers $a, b$, you can write $a = qb + r$ 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 $a$.

This repeated subtraction method is too 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".
Algorithm 4 Slow division (by repeated subtraction):

\[
q = 0 \\
r = a \\
\text{while } r \geq b \text{ do} \\
q \leftarrow q + 1 \\
r \leftarrow r - b \\
\text{end while}
\]

Example of long division

For example, 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)

\[
\begin{array}{c}
5 \\
\hline 723 | 41672542996 \\
3615 \\
\hline 552 \ldots \text{etc}
\end{array}
\]

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 \( 7 \times 5 < 41 \) whereas \( 7 \times 6 > 41 \). 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” and figure out how many times 723 goes into 5522, etc.

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 write down Java code and make it work.

Computational Complexity of Grade School Algorithms

For the case of \( N \) digit numbers, I only wrote down explicit algorithms for the case of addition and multiplication. So, let’s briefly compare the addition and multiplication algorithms in terms of the number of operations required. The addition algorithm has a \texttt{for} loop which is run \( N \) times. For each pass through the loop, there is a fixed number of simple operations. These operations include \( \leftarrow, \%, \div, + \). At this stage of your CS education, you don’t yet have the tools to say how expensive each of these operations are, i.e. how much time is required for each. (If fact, there are a lot of subtleties in specifying this.) So let’s just bundle all of them together for each pass through the loop and say that each pass takes \( c_2 \) units of time.

There are also a few operations in Algorithm 2 that are performed outside the \texttt{for} loop. We would say that the addition algorithm requires \( c_1 + c_2N \) operations, i.e. a constant \( c_1 \) plus a term that is proportional (with factor \( c_2 \)) to the number \( N \) of digits.
What about the multiplication algorithm? 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 the same basic operations performed as in the addition algorithm, namely \( \leftarrow, \%, /, + \). There is also the single digit multiplication operation \( * \).

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 \( c_5 + c_4 N + c_3 N^2 \). The same argument would apply for the second step of the multiplication algorithm, since again we have two nested for loops.

Keeping track of all these constants can give us a headache, and besides, we don’t know what these constants mean because they aggregate many different basic operations which we don’t know how to compare. Assigning a value to a variable, or computing the address of a particular array element \( r[i] \) or computing a single digital multiplication all take some unknown time. So again, its useless to talk about these constants in a meaningful way at this point.

What can we do? In computer science, we talk about how long algorithms take to run by ignoring the constants. Instead, we focus on a variable. In the case of the algorithms we’ve been discussing today, the variable is the number of digits \( N \) in the numbers that we are adding, subtracting, etc. We will say that addition takes time \( O(N) \), meaning that the time is dominated by \( N \). Sure there are some constants around, but we don’t care. Multiplication takes time \( O(N^2) \). The time it takes to perform multiplication also depends on \( N \) and there may be some constant amount of work that is independent of \( N \). But we don’t care about anything the dominant effect. If \( N \) is big, then the part that depends on \( N^2 \) is going to be our biggest concern.

**Exercises**

*For most lecture topics I will have a separate PDF for the Exercises. Here I put the exercises in the lecture notes because I discussed them in class.*

In the lecture, I asked you to consider the slow multiplication problem in which computes \( a \times b \) by adding \( a \) to itself \( b \) times, i.e. \( a + a + a + \ldots + a \). How does the time depend on the number of digits \( N \) of \( a \) and \( b \). (We assume they have the same number.)

I claimed the answer is \( O(N \times 10^N) \). Why? The algorithm is involves a for loop, where we loop \( b \) times. Since \( b \) has \( N \) digits, we consider the worst case that \( b \) could be any number up to \( 10^N \). So we are looping \( 10^N \) times. In each pass through the loop, we add the number \( a \) to the accumulated total. The number \( a \) has \( N \) digits, and so adding \( a \) involves \( N \) steps, i.e. addition takes time \( O(N) \). So this gives a total of \( O(N \times 10^N) \). Note that if \( N \) is large, say 80, then this is an astronomically large number.

Finally, I asked you to consider how grade school multiplication can be done so quickly, i.e. \( O(N^2) \). What trick does it use? The answer is that it uses the representation of numbers in terms of powers of 10. If we are given the number \( a \) in its decimal representation, then we can compute \( 10 \times a \) or \( 100 \times a \) etc very easily. We don’t need to add \( a \) to itself 10 times, or 100 times, etc. Your grade school teacher probably explained that to you, but you probably didn’t appreciate it. Perhaps you do now?