

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:

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

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 $a1$ and $a2$ and let's say they have N digits each. Then the two numbers can be represented as an array of single digit numbers $a1[]$ and $a2[]$. 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 $a1$ and $a2$ which are represented as arrays of digits

```

carry = 0
for i = 0 to N - 1 do
  r[i] ← (a1[i] + a2[i] + carry) % 10
  carry ← (a1[i] + a2[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 $a1$ multiplied by the single digits of the second number $a2$ (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 (table) 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 $a1$ and $a2$

```

for  $j = 0$  to  $N - 1$  do
   $carry \leftarrow 0$ 
  for  $i = 0$  to  $N - 1$  do
     $prod \leftarrow (a1[i] * a2[j] + carry)$ 
     $table[j][i + j] \leftarrow prod \% 10$ 
     $carry \leftarrow prod / 10$ 
  end for
   $table[j][N + j] \leftarrow carry$ 
end for
for  $i = 0$  to  $2 * N - 1$  do
   $sum \leftarrow carry$ 
  for  $j = 0$  to  $N - 1$  do
     $sum \leftarrow sum + table[j][i]$ 
  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 $c_1 + c_2N$ 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 *table*. 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.*

Binary numbers

The reason humans represent numbers using decimal (the ten digits from 0,1, ... 9) is that we have ten fingers. There is no other reason for this. There is nothing special otherwise about the number ten. Computers don't represent numbers using decimal. Instead, they represent numbers using binary. Let's make sure we understand what binary representations of numbers are. We'll start with positive integers.

In decimal, we write numbers using *digits* $\{0, 1, \dots, 9\}$, in particular, as sums of powers of ten. For example,

$$238_{ten} = 2 * 10^2 + 3 * 10^1 + 8 * 10^0$$

In binary, we represent numbers using *bits* $\{0, 1\}$, in particular, as a sum of powers of two:

$$11010_{two} = 1 * 2^4 + 1 * 2^3 + 0 * 2^2 + 1 * 2^1 + 0 * 2^0$$

I have put little subscripts (*ten* and *two*) to indicate that we are using a particular representation (decimal or binary). We don't need to always put this subscript in, but sometimes it helps.

It is trivial to write a decimal number as a sum of powers of ten and it is also trivial to write a binary number as a sum of powers of two, namely, just as I did above. So let's do something non-trivial, namely convert from binary to decimal and vice-versa.

To convert from a binary number to a decimal number, you need to know the decimal representation of the various powers of two.

$$2^0 = 1, 2^1 = 2, 2^2 = 4, 2^3 = 8, 2^4 = 16, 2^5 = 32, 2^6 = 64, 2^7 = 128, 2^8 = 256, 2^9 = 512, 2^{10} = 1024, \dots$$

Then, for any binary number, you write each of its bits as a power of 2 (in decimal) and then you add up these decimal numbers, e.g.

$$11010_{two} = 16 + 8 + 2 = 26.$$

The other direction is more challenging. How do you convert from a decimal number to a binary number? Next class, we will look at an algorithm for solving this problem.