Algorithms you learned long ago (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. For example, you learned how to add two numbers (say 3 and 4) by incrementing a "finger counter". The algorithm was something like this:

```
//   compute  firstnumber + secondnumber
//
count = firstNumber;
for i = 1 to secondNumber{
  count = count + 1;
}
```

Of course you didn't think of it this way! But this is basically what you did. And soon you memorized these single digit sums (4 + 7 = 11, etc).

Later on in grade school, you learned an algorithm 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:

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

What is 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 array `r[ ]`.

```
//  algorithm for adding two N digit numbers a[] and b[]
//
//      a[N-1]  .. a[0]
//    + b[N-1]  .. b[0]
//    ----------------
//  r[N]r[N-1] ..  r[0]
//
carry = 0;
for i = 0 to N-1 {
  r[i]   = (a[i] + b[i] + carry) mod 10;
  carry = (a[i] + b[i] + carry) / 10;       //  either 0 or 1
}
r[i+1] = carry;
```

This algorithm requires that you can compute the sum of two single digit numbers with + operator, and also (possibly) add 1 to that result.
[ASIDE: You also learned an algorithm for subtraction, which involved "borrowing" from `a[i+1]` in the case that `a[i] < b[i]`. I will spare you the details here. The point here is that you had to learn an algorithm for doing this.]

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 the following algorithm for multiplying a pair of `N` digit numbers. Notice that this algorithm can only be run if you know how to multiply two single digit numbers.

```
//  Algorithm for multiplying two N digit numbers
//
//      a[N-1]  .. a[0]
//    * b[N-1]  .. b[0]
//    ----------------
//

for j = 0 to N-1 {                // b index
  carry = 0;
  for i = 0 to N-1 {              // a index
    prod            = (a[i]* b[j] + carry);
    table[j][i + j] = prod mod 10;
    carry           = prod   / 10;       //  either 0 or 1
  }
  table[i][i+j+1] = carry;
}
carry = 0;
for i = 0 to 2*N-1 {
  sum = carry;
  for j = 0 to N-1 {
    sum = sum + table[j][i];
  }
  r[i] = sum mod 10;
  carry = sum / 10;
}
r[2*N] = carry;
```

Analysis of Algorithms

Let's compare the addition and multiplication algorithms. Both assume we have access to a lookup table that performs simple operations only, namely single digit addition and/or multiplication, truncated division, and `mod 10`. (These operations are so simple that they can be taught to children.)

The question we ask is, how many operations are required by each algorithm? 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 '`c1 + c2 N`' operations, i.e. a constant, plus a term that is propoortional to the number `N` of digits. Note that we do not distinguish the different simple operations here. Each counts for one "unit".

The multiplication involves two steps, each having a pair of `for` loops, one inside the other. This "nesting" of loops leads to $\texttt{N}^2$ passes through the operations within the inner loop. For each pass, there are various basic operations performed, namely table lookups for single digit multiplication and additions.

Suppose we consider the first step, in which we produce the two dimensional `table`. Suppose some number (say `c3`) of operations are inside both `for` loops, some number (say `c4`) of operations are inside just one of the `for` loops, and some number (say `c5`) of operations are outside both `for loops`. Then the number of operations is `c5 + c4 N + c3 N*N`.

The same argument would apply for the second step, except that the operations within both loops are now run `2 N*N` times. Note that `2` is just a constant which would be absorbed into the constant associated with the `N*N` term.

The key point to note is that the actual 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`. Because multiplication has an `N*N` term, whereas addition does not, the two algorithms behave quite differently when `N` is large, in particular, multiplication takes more operations when `N` is large.

From decimal to binary

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 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* $\{0, 1, \ldots, 9\}$, in particular, as sums of powers of ten. For example,

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

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.

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

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.

$$11010_{two} \ = \ 16 + 8 + 2$$

The other direction is more challenging. How do you convert from a decimal number to a binary number? Here is an algorithm and an example.

Algorithm

```
//      convert a positive integer m to binary
i = 0;
while (m > 0){
   b[i] = m % 2;      //   result is 1 if m is odd and 0 if m is even
   m    = m / 2;      //   truncated i.e.  "floor"
}
```

Example

remainders. For example,

i	m	b[ ]
0	$\boxed{241}$	
1	120	1
2	60	0
3	30	0
4	15	0
5	7	1
6	3	1
7	1	1
8	0	1
9	0	0
10	0	0
11	:	:

Next class we will discuss why this works.