Note 1: Addition, Multiplication Efficiency and Correctness

Kevin Moy

May 2020

- **Factoring**: Express N as a product of prime factors.
- **Primality**: Determine if N is a prime.

Factoring is **hard**: the fastest known methods for factoring N take time *exponential* to the number of bits of N.However, *testing* primality of N is efficient. Interestingly, this strange disparity forms the core of secure communication technology!

We need to develop algorithms for many types of numerical computation. Let's start with basic arithmetic.

1 Addition

Let's go back to elementary school addition.

We know, internally or externally, that **for any base, the sum of any three single-digit numbers is at most two digits long.** Decimal: 9+9+9 = 27, a 2 digit number. In binary, the maximum sum of 3 single digit numbers is 1+1+1=3, which is 11_2 = a 2 digit number in binary.

So adding in any base just consists of aligning right ends of numbers and then perform a single right-to-left pass in which the sum is computed digit by digit, maintaining the overflow as a carry. Even with a carry from overflow, this results in at most 3 single-digit additions, so at any given step, three single-digit numbers are added.

Let's analyze efficiency of this algorithm. Given two binary numbers x and y, how long does it take to add them via this standard algorithm? We express this as a function of the input size, in this case the number of bits of x and y. If we suppose x and y are n bits long, then $x + y$ is at most $n + 1$ bits. Each bit is computed in constant time. Thus running time is in form $c_1 n + c_0$ where c_0, c_1 are constants- this is linear time. **So addition of 2 numbers takes $O(n)$ time.**

Is there something better? I.e. a faster algorithm than this? For addition the answer is no: it takes more than n operations to just write down the addends.

Isn't binary addition a single instruction on computers? Yes- if addends are within *word length* (32 or 64 bits) of computers. However, what about addends with thousands of bits? For these kinds of numbers, computers basically do bit-by-bit addition. It's also just a good idea in general to study even simple algorithms to build intuition. The **bit complexity** of an algorithm is the general number of ops on individual bits—which reflects the amount of hardware, transistors, wires, etc. necessary for implementing the algorithm in a computer.

2 Multiplication and Division

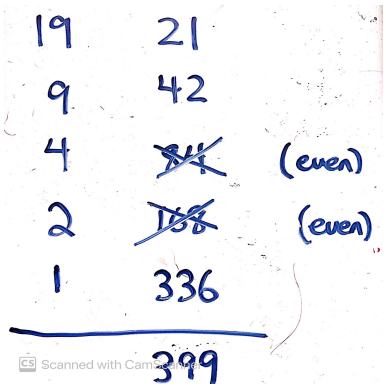
Elementary-school multiplication of x and y consists of creating an array of intermediate sums, ($x *$ a single digit of y), right to left again. We left-shift each sum then add them all for the final product. Binary multiplication is the same exact process as base 10- we won't bother showing an example.

But notice that in binary, since each digit of y is either 1 or 0, each intermediate sum must be 0 or x! Left-shifting is the same as multiplying the intermediate sum by the base (by 2 in binary).

Let's analyze efficiency of the standard multiplication algorithm. If x and y are n bits, we have n intermediate rows with numbers up to $2n$ bits (implicit 0 bits caused by left-shifting). For n rows, going 2 at a time, we have $n-1$ additions, each addition being $O(n)$. **So multiplication of 2 numbers takes $O(n^2)$ time**. Slower than addition, of course.

2.1 Al Khwarizmi's multiplication algorithm

This dude named Al Khwarizmi had another multiplication algorithm: To multiply two *decimal* numbers x and y, write them next to each other. Floor divide the first number by 2, and double the second number. Repeat this till the first number gets down to 1. Then strike out all the rows in which the first number is even, and add up whatever remains in the second column. Below is an example on 19 * 21:



However, notice that this dude's algorithm is actually just the same exact thing as binary multiplication, just in disguise. Notice we add 21 + 42 + 336, the exact same intermediate sums we add in binary multiplication, i.e. the multiples of 21 by powers of 2. What we did with 19 is floor divide it consistently by 2- which actually gives us its binary representation. An interesting relation of base-10 to binary multiplication!

Let's look at the algorithmic **pseudocode**:

```
function multiply(x,y):
    if y = 0: return 0
    z = multiply(x, ⌊y/2⌋)
    if y is even:
        return 2z
    else:
        return x + 2z
```

(Note that in 19*21 visual example x is halved each step- it doesn't really matter which one is halved and which is doubled, since 19*21 = 21*19).

We can also represent this algorithm in piecewise function form:

$$x * y = \begin{cases} 2(x * \lfloor y/2 \rfloor) & \text{if y even} \\ x + 2(x * \lfloor y/2 \rfloor) & \text{if y odd} \end{cases}$$

Now let's prove algorithm correctness. Here, checking correctness is just verifying that it mimics the rule and that it handles the base case (y = 0) properly. We see it's easily correct.

What about runtime? We halve y (right-shift y) at each step, so we have n recursive calls. In each recursive call:

- Division by 2 (right shift) - $O(n)$ time
- Odd/even test (looking up last bit)- constant time
- Possible Multiplication by 2 (left shift) - $O(n)$ time

- Possible Addition- $O(n)$ time

So Al-Khwarizmi's algorithm takes $O(n^2)$ time, just as before.

3 A Parting Thought

We know we can't do better than $O(n)$ for addition. But can we do better than $O(n^2)$ for multiplication? Multiplication is adding n multiples of an input, so we can't right?

We'll see in our next note that we actually can.