Lecture II. Numbers and Arithmetic

Whenever we talk about complexity of an algorithm it is important that we keep in mind what exactly is the input to our algorithm. In classical computational theory big-O is usually a function of the total length of the input. While this is useful when working with a Turing machine, every element of which is just a bit, this notation would become cumbersome for many of the problems that we will discuss in this class. For example when we talk about algorithms we will usually assume that adding, subtracting, multiplying, or dividing two numbers can be accomplished in O(1) time. This is almost always true in practice – the numbers within a computer are usually represented as a binary string of up to 64 bits. Arithmetic operation are implemented efficiently "in hardware", and the results will be available just one clock cycle later. When implementing most algorithms this time will rarely be a concern. However fixed-size numbers can only represent a limited range of values. In particular, the way 64 bit integers are implemented on most machines one can represent whole numbers in the range $[-2^{63} - 1, 2^{63}]$. If you want to use bigger numbers, you will have to implement them explicitly, and to re-implement for these all of the arithmetic operations (of course you can also Google-around for a ready implementations of BigIntegers, which are part of standard libraries for many languages, such as Java and Python, but what would be the fun in that?). Why would you want to implement very large numbers? One prominent example is Cryptography (see Chapter 1 of the book). Many cryptographic schemes depend on non-invertible functions: those which are easy to compute, but hard to invert (i.e. given the answer, find the argument). One prominent example of such a function is factoring. Given two prime numbers, *a* and *b*, computing their product $c = a * b$ is relatively easy (as we will soon see). However given a number *c* that is a product of two primes, how would one find out what *a* and *b* where? You may suggest the following algorithm:

```
int factor (int c) {
   for (i = 0; i < c; i++) {
       if (c % i == 0)
          return i; // returns the smallest of the two factors
   }
}
```

What's so difficult about this, you may ask. This algorithm runs in *linear* time, it is quite fast. This is not true. This algorithm runs in time O(c), which is in fact *exponential* in terms of the size of *c,* if *c* is representedusing binary (or any other base besides unary): only 64 bits can encode all numbers up to 2^{64}. So this algorithm is, in fact, *exponential* in the **size of** *c* (which we will write as $|c|$). In fact, the popular RSA cryptosystem uses 1024 bit integers. Because no known fast algorithms exist for factoring such large numbers you can securely submit your credit card information to a website.

Well, if factoring is so slow, you may ask, what about the other operations? Can we multiply two large numbers quickly, in order to get the product in the first place? To understand how to do this, we first need to decide how to represent the numbers on a computer (we will call these BigInts). We will use the binary system, with every digit (bit, to be technically correct) stored as an element of an array A. A[0] will be the lowest order bit (2^0), while A[k], is the 2^kth bit. (If you have difficulty with the positional notation system we recommend the following link: http://www.youtube.com/watch?v=a81YvrV7Vv8.) For simplicity, assume that A.size will tell us the largest bit that is not zero in the array A. Let us now define a few simple operations on numbers which are represented in this way. Let us start with addition. We will do this exactly how you learned to do this in grade school – write the numbers one under the other, add the two rightmost digits. If you get a value that is greater than a single digit (9 in grade school ,and 1 in our case) you carry a one. And so, on now also using the carry. This algorithm becomes the following piece of code:

```
BigInt add (BigInt a, BigInt b) {
   int carry = 0;
   int i = 0;
   BigInt c;
   for (i=0; i < MAX(a.size, b.size); i++) {
      c[i] = (a[i] + b[i] + carry) % 2;
      if  (a[i] + b[i] + carry > 1)
        carry = 1;
      else carry =  0;
   }
   return c;
}
```

This algorithm runs in linear time (in the size of the two integers, not, their values). Subtraction can be implemented in a similar manner. Next is multiplication. In the multiplication method from grade school, you multiplied one number by each digit of the other, shifting the result by one for each subsequent digit, and then added the results together. For binary this is easy: there are only two bits: Multiplying by one is just the original number, while by zero gives you zero. Adding we have already implemented. The only remaining step is shifting (we will pad the strings with zeroes on the right):

```
BigInt lshift(BigInt a, int k) { // returns a shifted left by k bits
   BigInt c;
   int i;
   for (i = a.size-1; i >= 0 ; i--) {
      c[i+k] = a[i];
   }
   for (i = k-1; i >= 0; i--) {
      c[i] = 0;
   }
   return c;
}
```

So now we can build the following algorithm:

```
BigInt multiply (BigInt a, BigInt b) {
   BigInt c;  //initialized to all zeroes
   int i;
   for (i=0; i < a.size; i++) {
      if (a[i] != 0) {  //if it is zero, nothing to do
        c = add (c, lshift (b, i));
      }
   }
   return c;
}
```

What is the running time of the resulting algorithm? We do O(|a|) passes through the loop, with each requiring O(|b|) time (addition and shifts). So the overall runtime is $O(n^2)$ if $n = |a| = |b|$. Can we do better? Well, in some special cases, certainly. For example, multiplication (and division) by 2, or any of its powers, can be implemented using shift operations. Just as multiplying by 10 in decimal adds a zero on the right, the same happens when multiplying by 2 in binary. However what about in the other cases? We were able to make a faster exponentiation algorithm by using a little bit of algebra, so perhaps we can do the same here. We can try to halve one of the numbers at every step:

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

However halving a number only reduces the number of bits in it by one. So the recursion will have number of levels proportional to |b|. At each one we will do O(n) work, so this approach will not give us a speedup. Instead of dividing the number by two, we have to divide it into two parts, the left and the right. Let us start by recalling that if we split a number into two equal parts (call them a_l and a_r), the original number is $2^{n/2}*a_l + a_r$. So the multiplication a*b can be rewritten as

$$(2^{n/2}*a_l + a_r)* (2^{n/2}*b_l + b_r) = 2^n a_l b_l + 2^{n/2}(a_l b_r + b_l a_r) + a_r b_r$$

If we would compute this directly, we actually would get a recurrence of $T(n) = 4\,T(n/2) + O(n)$, which means to compute the result for n, we will do four sub-jobs of size n/2, and then spend O(n) time merging the results (additions and multiplying by powers of 2). What will the running time be? Our recursion will have O(log n) levels, but the amount of work will double to every next level: n at the top level, $4*n/2 = n * 2$ at the next level, etc. Only the bottom level will have $n * 2^{\log (n)} = n^2$ operations! While at first this may seem like a *slower* way of computing the product of two numbers, in fact it turns out to be equivalent. If we sum up all of the levels, $n^2 + n^2/2 + n^2/4 + \ldots < 2*n^2$, so the overall running time of our algorithm is the same. However let us go back to the equations, and try to use a little bit more algebra to solve them in a smarter way. Note that

$$(a+b)*(c+d) = ac + bc + ad + bd$$
$$bc + ad = (a+b)*(c+d) - ac - bd$$

This suggests that the $a_l b_r + b_l a_r$ of the original equation can be computed using the result of $(a_l+a_r)*(b_l+b_r)$, $a_l b_l$, and $a_r b_r$, and that the four multiplications can be rewritten with three:

```
BigInt multiply (BigInt a, BigInt b) {
   BigInt c, al, ar, bl, br, p1, p2, p3;
   if (n = 1) return a*b;  //assume that n is the size of the two integers
   al = leftside(a); ar = rightside(a);
   bl = leftside(b); br = rightside(b);
   p1 = multiply(al, bl);
   p3 = multiply(ar, br);
   c = multiply(add(al,ar),add(bl,br));
   p2 = subtract (subtract(c, p1), p2); // c – p1 – p3
   return add(add(lshift(p1, n) + lshift(p2, rshift(n,1))), p3)
}
```

What is the running time of this? Well, we now only have 3 recursive calls, while the amount of work at each level has not changed (at least from a big-O perspective). Now each level will have 1.5 times more work then the previous one, with the bottom level having $n * 1.5^{\log(n)} = n^{1.59}$. Again, the overall big-O running time will be equal to the bottom level: $n^{1.59} + n^{1.59} * 2/3 + n^{1.59} * 4/9 \ldots < 5/2 * n^{1.59}$

This gives us an $O(n^{1.59})$ algorithm for multiplying two large numbers. In fact, faster algorithms do exist, they run in almost linear time, but they are based on very similar ideas. You may ask about the practical aspects of such an algorithm: given all of the other operations that need to happen, will it actually be faster? The answer is *only if the numbers are big.* The large constants, plus the overhead of the recursion will make the algorithm slower for smaller numbers, and even the theoretical dvantage of $n^{.41}$ is not that big: $100^{0.41} = 6.3$; $1000^{0.41} = 15.8$. However if the numbers are *very* big, this will be faster. Of course the natural modification is to only use divide and conquer for the bigger numbers, until they are small enough that the $O(n^2)$ algorithm becomes faster.