Notes for Lecture 2

1 Integer Multiplication

The standard multiplication algorithm takes time $\Theta(n^2)$ to multiply together two n digit numbers. This algorithm is so natural that we may think that no algorithm could be better. Here, we will show that better algorithms exist (at least in terms of asymptotic behavior).

Imagine splitting each number x and y into two parts: $x = 2^{n/2}a + b, y = 2^{n/2}c + d$. Then:

$$xy = 2^n ac + 2^{n/2}(ad + bc) + bd.$$

The additions and the multiplications by powers of 2 (which are just shifts!) can all be done in linear time. So the algorithm is

```
function Mul(x,y)
   n = common bit length of x and y
   if n small enough
      return x*y
   else
      write x = 2^(n/2)a+b and y = 2^(n/2)*c+d       ... no cost for this
      p1 = Mul(a,c)
      p2 = Mul(a,d) + Mul(b,c)
      p3 = Mul(b,d)
      return 2^n*p1 + 2^(n/2)*p2 + p3
   end
```

We have therefore reduced our multiplication into four smaller multiplications problems, so the recurrence for the time $T(n)$ to multiply two n-digit numbers becomes

$$T(n) = 4T(n/2) + \Theta(n).$$

Unfortunately, when we solve this recurrence, the running time is still $\Theta(n^2)$, so it seems that we have not gained anything. (Again, we explain the solution of this recurrence later.)

The key thing to notice here is that four multiplications is too many. Can we somehow reduce it to three? It may not look like it is possible, but it is using a simple trick. The trick is that *we do not need to compute ad and bc separately; we only need their sum $ad + bc$.* Now note that

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

So if we calculate ac,bd, and $(a + b)(c + d)$, we can compute $ad + bc$ by the subtracting the first two terms from the third! Of course, we have to perform more additions, but since the bottleneck to speeding up the algorithm was the number of multiplications, that does not matter. This means the 3 lines computing `p1`, `p2`, and `p3` above should be replaced by

```
p1 = Mul(a,c)
p3 = Mul(b,d)
p2 = Mul(a+b,c+d) - p1 - p3
```

The recurrence for $T(n)$ is now

$$T(n) = 3T(n/2) + \Theta(n),$$

and we find that $T(n)$ is $\Theta(n^{\log_2 3})$ or approximately $\Theta(n^{1.59})$, improving on the quadratic algorithm (later we will show how we solved this recurrence).

If one were to implement this algorithm, it would probably be best not to divide the numbers down to one digit. The conventional algorithm, because it uses fewer additions, is more efficient for small values of n. Moreover, on a computer, there would be no reason to continue dividing once the length n is so small that the multiplication can be done in one standard machine multiplication operation!

It also turns out that using a more complicated algorithm (based on a similar idea, but with each integer divided into many more parts) the asymptotic time for multiplication can be made very close to linear – $O(n \cdot \log n \cdot \log\log n)$ (Schönhage and Strassen, 1971). Note that this can also be written $O(n^{1+\epsilon})$ for any $\epsilon > 0$.

2 Solving Divide-and Conquer Recurrences

A typical divide-and-conquer recurrence is of the following form:

$$T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n),$$

where we assume that $T(1) = O(1)$. This corresponds to a recursive algorithm that in order to solve an instance of size n, generates and recursively solves a instances of size n/b, and then takes $f(n)$ time to recover a solution for the original instance from the solutions for the smaller instances. For simplicity in the analysis, we further assume that n is a power of b: $n = b^m$, where of course m is $\log_b n$. This allows us to sweep under the rug the question "what happens if n is not exactly divisible by b" either in the first or subsequent recursive calls. We can always "round n up" to the next power of b to make sure this is the case (but, of course, in a practical implementation we would have to take care of the case in which the problem would have to be split in slightly unbalanced subproblems).

Certainly, we have $T(n) \geq f(n)$, since $f(n)$ is just the time that it takes to "combine" the solutions at the top level of the recursion.

Let us now consider the number of distinct computations that are recursively generated. At each level that we go down the recursion tree, the size of instance is divided by b, so the recursion tree has m levels, where $m = \log_b n$. For each instance in the recursion tree, a instances are generated at the lower level, which means that $a^m = a^{\log_b n} = n^{\log_b a}$ instances of size 1 are generated while solving an instance of size n. Clearly, $n^{\log_b a}$ is also a lower bound for the running time $T(n)$.

Indeed, in most cases, $n^{\log_b a}$ or $f(n)$ can also be tight upper bound (up to a multiplicative constant) for $T(n)$.

Specifically:

- If there is an $\epsilon > 0$ such that $f(n) = O(n^{\log_b a - \epsilon})$, then $T(n) = \Theta(n^{\log_b a})$.
- If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a})$.
- If there is an $\epsilon > 0$ such that $f(n) = \Omega(n^{\log_b a + \epsilon})$, then $T(n) = \Theta(f(n))$.

CLR calls this the "Master Theorem" (section 4.3 of CLR). The Master theorem applies if $f(n)$ are exactly of the same order of magnitude, or if their ratio grows at least like n^ϵ, for some $\epsilon > 0$, but there are possible values for a, b and $f(n)$ such that neither case applies.