DISCLAIMER: These notes are not necessarily an accurate representation of what I said during the class. They are mostly what I intend to say, and have not been carefully edited.

- Main topics:
  - Faster Multiplication via Div-Conquer.
  - Recursion Trees, ideas on how to prove master theorem.

- From Last time:
  - Distinction between \( f(n) \), \( O(n) \), \( T(n) \):
    1. \( O(\cdot) \) is special, use it to bound runtimes / compare functions.
    2. \( f(n) \) is a function of \( n \), \( T(n) \) is similar to it.
    3. \( O(n) \) is a short hand for \( O(g(n)) \) where \( g(n) = n \).
  - \( \log^a n \leq O(n^b) \).
    1. \( \log^a n \) denotes \((\log n)^a\), not applying \( \log a \) times to \( n \).
    2. Proving this is not critical, can use this,
    3. as well as \( n^a \leq O(b^n) \) for any \( b > 1 \).

Faster Multiplication Algorithm

We continue from the multiplication algorithm from last class, but improve its running time to sub-quadratic. The textbook attributes this trick to Gauss, although another common citation is to Karatsuba (see e.g. Wikipedia).

The plan is to reduce the number of recursive calls, specifically the number of multiplies involving. Note that the two middle ‘cross terms’ \( ad \) and \( bc \) both have coefficients \( 10^{n/2} \), and can be obtained from \((a + b) \times (c + d)\) along with \( ac \) and \( bd \):

\[
ad + bc = (a + b) \times (c + d) - ac - bd.
\]

\((a + b)\) is ‘simpler’ than \( a \times 10^{n/2} + b \) because both \( a \) and \( b \) have \( n/2 \)-digits, so their sum has at most \( n/2 + 1 \) digits. This is only half as many as the original number which has \( n \) digits. For the running example of multiplying 1234 with 5678, this gives:

\[
\begin{align*}
1200 \times 78 + 34 \times 5600 &= 100 \times (12 \times 78 + 34) \\
&= 100 \times [(12 + 34) \times (56 + 78) - 12 \times 56 - 34 \times 78]
\end{align*}
\]
Recall also that we need to compute $ac$ and $bd$ for the first $(ac \times 10^n)$ and last $(bd)$ terms. So this new step replaces computing $ad$ and $bc$ separately with computing $(a + b) \times (c + d)$ and some additions / subtractions involving quantities that we already computed.

Hence, it allows us to multiply two $n$-digit numbers through multiplying three pairs of $n/2$-digit numbers, along with some addition / subtraction operations. This gives the recurrence

$$T(n) = 3T(n/2) + O(n).$$

This is described by the master theorem with $a = 3$, $b = 2$, and $d = 1$. We have

$$\log_b a = \log_2 3 = 1.584963 \ldots > 1,$$

so we’re in case 3 again, giving

$$T(n) = O(n^{1.585}),$$

which is strictly better than the two quadratic time algorithms that we’ve discussed earlier. Note that this is the first example that we have where the master theorem gives one of the simplest ways of solving the runtime recurrence.

## 1 Recursion Trees

We now study the runtime recurrence

$$T(n) = aT\left(\frac{n}{b}\right) + O\left(n^d\right)$$

in a bit more detail.

When unrolled this says that to solve a problem of size $n$, the cost is the sum of:

1. $O(n^d)$,
2. the cost of solving $a$ copies of problems of size $n/b$.

For each of the problems of size $n/b$, the cost is once again:

1. $O((n/b)^d)$, 
2. the cost of solving $a$ copies of problems of size $n/b^2$.

This leads to a recursion tree where the nodes represent recursive calls, labeled with the size of the input instances being given. Figure 2.3. on page 59 of the textbook has a good example of this, and you can find more online by searching for ‘recursion trees’. In general the runtime recurrence implies:

1. Each node leads to $a$ more recursive calls,
2. each of such calls get smaller by a factor of $b$. 

This continues until we get to instances with size at most $b$, which we treat as a constant.

It then makes sense to evaluate the total cost incurred by level $k$ of this recursion tree. At that point the problem has decreased by a factor of $b^k$, while the number of calls is $a^k$. Therefore the total cost is:

$$a^k \times O \left( \frac{n}{b^k} \right)^d = O(n^d) \times \left( \frac{a}{b^d} \right)^k. \quad (4)$$

So everything depends on the ratio

$$\frac{a}{b^d},$$

and this is where our three cases for Master theorem come from:

1. If $a/b^d < 1$, then the first term gives the total, $n^d$.

2. If it’s exactly 1, then each term is the same, so we get a total equaling to $O(n^d)$ times the number of terms. Since each step decreases the size by a factor of $b$, this number of steps is $\log_b n$, which as $b$ is treated as a constant, gives $O(\log n)$.

3. If it’s greater than 1, then the last term dominates, and we get a total of (ignoring constants):

$$n^d \left( \frac{a}{b^d} \right)^{\log_b n} = n^{\log_b a}.\,$$

The one other thing that we ignored is using $\lceil n/b \rceil$ in place of $n/b$. The effect of this depends on the value of $b$, but it can be checked that it increases the size of any recursive call by at most 1. This is then absorbed into the $O(\cdot)$. 

3