CSE 326 Autumn 2006

Section 2 Notes

I glossed over some calculation details and also made some flat-out mistakes in section. Here is a (more) correct version of the notes. Sorry for the confusion.

First some useful identities and sums to know for solving recurrences:

$$c \log_b a = \log_b a^c \tag{1}$$
$$b^{\log_b a} = a \tag{2}$$
$$c^{\log_b a} = a^{\log_b c} \tag{3}$$
$$\sum_{i=0}^{n} r^i = \frac{r^{n+1} - 1}{r - 1} \tag{4}$$

If you are interested in the proofs, e-mail me.

1. Example recurrences

   It isn't really important to understand the details of multiplication algorithms. We just solve the recurrences below:

   Note that the analysis below is sloppy with respect to rounding logarithms to the nearest integer, but it does not change the asymptotic bounds.

   (a) 3rd grade algorithm

   The naive 3rd grade recursive algorithm divides numbers into 4 parts and does 4 multiplications, giving us the following recurrence:

   $$T(n) = 4T(\frac{n}{2}) + O(n) \tag{5}$$

   We can expand the term $T(\frac{n}{2})$ by substituting $\frac{n}{2}$ in equation 5 wherever we see an n.

   $$\begin{aligned} T(n) &= 4[4T(\frac{n}{4}) + O(\frac{n}{2})] + O(n) & (6)\\ &= 4 \cdot 4[4T(\frac{n}{8}) + O(\frac{n}{4})] + 4O(\frac{n}{2}) + O(n) & (7) \end{aligned}$$

   How many times can we divide n by 2 before we get to the base case, $T(1) = O(1)$? That is, how many times can we expand the recurrence? We will call this number k below.

   $$\frac{n}{2^k} = 1 \Rightarrow n = 2^k \Rightarrow k = \log_2 n \tag{8}$$

   This is the definition of a logarithm, the inverse of exponentiation.

   Now we can express the recurrence as a sum:

   $$\begin{aligned} T(n) &= 4^{\log_2 n} T(1) + \sum_{i=0}^{\log_2 n} 4^i (\frac{1}{2})^i O(n) & (9)\\ &= 2^{2\log_2 n} + O(n) \sum_{i=0}^{\log_2 n} 2^i & (10) \end{aligned}$$

We can simpify the first term with identity 1 and the second term (which is a sum) with identity 4.

$$T(n) = 2^{\log_2 n^2} + O(n)\frac{2^{\log_2 n+1} - 1}{2-1} \tag{11}$$

We can apply identity 2 to both terms to simplify it further:

$$\begin{aligned} T(n) &= n^2 + O(n)(2n-1) \qquad (12)\\ &= O(n^2) \qquad (13) \end{aligned}$$

(b) Karatsuba's algorithm

The recurrence for Karatsuba's algorithm performs 3 multiplications on subparts instead of 4.

$$T(n) = 3T(\frac{n}{2}) + O(n) \tag{14}$$

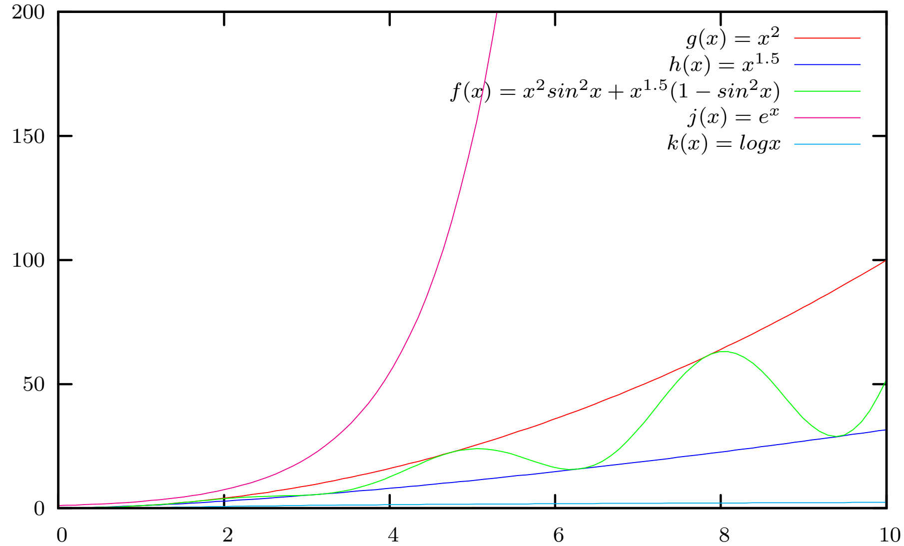
Again we expand the recurrence:

$$\begin{aligned} T(n) &= 3[3T(\frac{n}{4}) + O(\frac{n}{2})] + O(n) \qquad (15)\\ &= 3 \cdot 3[3T(\frac{n}{8}) + O(\frac{n}{4})] + 3O(\frac{n}{2}) + O(n) \qquad (16)\\ &= 3^{\log_2 n} + O(n)\sum_{i=0}^{\log_2 n}(3/2)^i \qquad (17)\\ &= n^{\log_2 3} + O(n)\frac{3^{\log_2 n+1}2^{-\log_2 n+1} - 1}{1/2} \qquad (18)\\ &= n^{\log_2 3} + O(n)\frac{3^{\log_2 n+1}2^{-\log_2 n+1} - 1}{1/2} \qquad (19)\\ &= n^{\log_2 3} + 2O(n)n^{\log_2 3+1}2^{\log_2 n^{-1}} - 1 \qquad (20)\\ &= n^{\log_2 3} + 2O(n)n^{\log_2 3+1}n^{-1} - 1 \qquad (21)\\ &= n^{\log_2 3} + 4n^{\log_2 3} - 1 \qquad (22)\\ &= O(n^{\log_2 3}) \qquad (23)\\ &\approx O(n^{1.585}) \qquad (24) \end{aligned}$$

Therefore, Karatsuba's algorithm is asymptotically faster than the 3rd-grade algorithm.

2. Asymptotic notation: Big-O and friends

Unfortunately, I reversed the definitions of $O(n)$ with $\Omega(n)$ and likewise of $o(n)$ with $\omega(n)$. Ack! I hang my head in shame. Thanks to Joanna Salacka for pointing this out. Here is the diagram again and the definition of the functions:



Here are the correct asymptotic relationships:

$$\begin{aligned} f(x) \in o(j(x)) &\Rightarrow f(x) \in O(j(x)) \\ f(x) \in \omega(k(x)) &\Rightarrow f(x) \in \Omega(k(x)) \\ & f(x) \in O(g(x)) \\ & f(x) \in \Omega(h(x)) \end{aligned}$$

Note that $\sin^2 x$ varies between 0 and 1, and that by definition $f(x)$ is tightly lower-bounded by $x^{1.5}$ and tightly upper-bounded by x^2. I did this to be tricky in the following ways.

The upper bound function x^2 is in $\omega(x^{1.5})$, So, x^2 will always intersect the lower bound of $f(x)$ no matter how small of a constant we multiply it with. So $g(x) = x^2$ will always grow faster than $f(x)$'s lower bound but there exists some (small) constants c for which $cg(x)$ cannot grow as fast as $f(x)$'s upper bound. For these constants, $f(x)$ and $cg(x)$ intersect an infinite number of times, and therefore, $f(x)$ is *not* in $o(x^2)$. However, it is still in $O(x^2)$ because there are other (large) constants c for which $f(x) \leq cg(x)$.

Likewise, there are (small) constants c for which $f(x) \geq ch(x)$, so that's why $f(x) \in \Omega(h(x))$. However, there are also some (large) constants c for which $ch(x)$ can grow faster than $f(x)$'s lower bound, and again we will have an infinite number of intersections between the two functions. Therefore, $f(x)$ is not in $\omega(h(x))$.

In reality, most algorithm running times are polynomials, and you will never see trigonometric functions like sine. I'll make future examples simpler and more realistic.

3. Merge sort

I made a mistake in both sections about the arguments to mergesort. The algorithm should take a beginning and ending index, and the midpoint should be defined as their average (rounded to an integer).

Here is the correct pseudocode for merge sort.

```
global arrays x[m], y[m]

MergeSort(start, end)

  if (start == end)
    return

  midpoint = floor((end + start)/2)
  MergeSort(start, midpoint)
  MergeSort(midpoint + 1, end)
  i = start, j = midpoint + 1, k = 1

  while (i < midpoint or j < end)
    if (x[i] < x[j])
      y[k] = x[i], i++, k++
    else
      y[k] = x[j], j++, k++

  copy y[1 to (end-start)] to x[start to end]
  return
```

Here is the recurrence for mergesort:

$$T(n) = 2T(\frac{n}{2}) + O(n) \tag{25}$$

And here is how to solve it:

$$\begin{aligned}
T(n) &= 2[2T(\frac{n}{4}) + O(\frac{n}{2})] + O(n) && (26)\\
&= 2 \cdot 2[2T(\frac{n}{8}) + O(\frac{n}{4})] + 2O(\frac{n}{2}) + O(n) && (27)\\
&= 2^{\log_2 n} + O(n) \sum_{i=0}^{\log_2 n} 1 && (28)\\
&= n + O(n) \log_2 n && (29)\\
&= O(n \log_2 n) && (30)
\end{aligned}$$

Therefore, mergesort is asymptotically faster than insertion or selection sort, which both take $O(n^2)$ time.