CS170–Fall 2012 — Solutions to Homework 1

Ben Augarten, section 106, `cs170-bo`

August 31, 2012

1. For question 1, I assume the following basic relations:
   $n^c = \Omega(\log(n))$, $0 < c < 1$
   $n = \Omega(n^c)$, $0 < c < 1$
   $n = O(n^c)$, $c > 1$

   (a) $n - 100 = \Theta(n - 200)$, in the limits as $n \to \infty$, both act like n, $\lim_{n\to\infty} \frac{f}{g} = 1$

   (b)
   $$\begin{aligned} &\lim_{n\to\infty} \frac{n^{1/2}}{n^{2/3}} \\ &\lim_{n\to\infty} n^{-1/3} = 0 \\ &n^{1/2} = O(n^{2/3}) \end{aligned}$$

   (c)
   $$\begin{aligned} &\frac{100n + \log(n)}{n + \log(n)^2} \\ &\lim_{n\to\infty} \approx \frac{100n}{n} = 100 \\ &100n + \log(n) = \Theta(n + \log(n)^2) \end{aligned}$$

(d)

$$\begin{aligned}
&\frac{n\log(n)}{10n\log(10n)}\\
=&\frac{n\log(n)}{10n(\log(10)+\log(n))}\\
=&\frac{n\log(n)}{10n\log(n)+10\log(10)n} \qquad \text{(shows } \log(cn)=\Theta(\log(n))\text{)}\\
\lim_{n\to\infty}\approx&\frac{n\log(n)}{10n\log(n)}\\
=&10\\
n\log(n)=&\Theta(10n\log(10n))
\end{aligned}$$

(e) $\log(2n) = \Theta(\log(3n))$, by part (d)

(f) $\log(n^2) = 2 * \log(n)$
$10\log(n) = \Theta(\log(n^2))$

(g)

$$\begin{aligned}
\lim_{n\to\infty}\frac{n^{1.01}}{n\log^2(n)}\\
\lim_{n\to\infty}\frac{n^{.01}}{\log^2(n)}\\
\lim_{n\to\infty}\frac{n^{.005}}{\log(n)}&=\infty\\
n^{1.01}&=\Omega(n\log^2(n))
\end{aligned}$$

(h)

$$\begin{aligned}
\lim_{n\to\infty}\frac{\frac{n^2}{\log(n)}}{n\log^2(n)}\\
\lim_{n\to\infty}\frac{n}{\log^3(n)}&=\infty \text{ using L'Hopital's rule}\\
\frac{n^2}{\log(n)}&=\Omega(n\log^2(n))
\end{aligned}$$

(i)

$$\lim_{n\to\infty}\frac{n^{.1}}{\log(n)^{10}}=\infty \text{ same as h, keep using L'Hopital's rule}$$

$$n^{.1}=\Omega(\log(n)^{10})$$

(j)

$$\lim_{n\to\infty} \frac{\log(n)^{\log(n)}}{\frac{n}{\log(n)}}$$
$$\lim_{n\to\infty} \frac{\log(n)^{\log(n)+1}}{n}$$
$$\text{let } m = \log(n) \lim_{m\to\infty} \frac{m^{m+1}}{2^m} = \infty$$
$$\log(n)^{\log(n)} = \Omega(\frac{n}{\log(n)})$$

(k)

$$\lim_{n\to\infty} \frac{\sqrt{n}}{\log^3(n)} = \infty \text{ using L'Hopital's rule}$$
$$\sqrt{n} = \Omega(\log^3(n)$$

(l)

$$\lim_{n\to\infty} \frac{\sqrt{n}}{5^{\log_2(n)}}$$
$$\text{let } x = \log_2(n) \lim_{x\to\infty} \frac{2^{x/2}}{5^x} = 0$$
$$\sqrt{n} = O(5^{\log_2(n)})$$

(m)

$$\lim_{n\to\infty} \frac{n2^n}{3^n}$$
$$\lim_{n\to\infty} n * \left(\frac{2}{3}\right)^n = 0$$
$$n2^n = O(3^n)$$

(n)

$$\lim_{n\to\infty} \frac{2^n}{2^{n+1}} = \frac{1}{2}$$
$$2^n = \Theta(2^{n+1})$$

(o)

$$\lim_{n\to\infty} \frac{n!}{2^n}$$
$$\frac{1}{2} * \frac{2}{2} * \frac{3}{2} * * \frac{n-1}{2} * \frac{n}{2}$$

as n approaches infinity, this series must approach infinity because each successive term is increasing and all except the first are at least one (I don't know if this is a formal proof but I forget 1B, the following conclusion seems rather intuitive)
$n! = \Omega(2^n)$

(p)

$$\lim_{n\to\infty} \frac{\log(n)^{\log(n)}}{2^{\log^2(n)}}$$
$$\lim_{x\to\infty} \frac{x^x}{2^{x^2}} = 0 \text{ courtesy of wolframalpha}$$
$$\log(n)^{\log(n)} = O(2^{\log^2(n)})$$

(q)

$$\lim_{x\to\infty} \frac{\sum_{i=1}^{n} i^k}{n^{k+1}}$$
$$\lim_{x\to\infty} 1^k + 2^k + 3^k + \ + (n-2)^k + (n-1)^k + \frac{n^k}{n^{k+1}} = \infty$$
$$\sum_{i=1}^{n} i^k = \Omega(n^{k+1})$$

2. (a) $g(n) = 1 + c + c^2 + \; + c^n$, $c < 1$ This is a geometric series with $0 < c < 1$ (it converges).
   $\lim_{n\to\infty} \frac{g(n)}{1} = \frac{1+c+c^2++c^n}{1} = \frac{1}{1-c}$, a constant so $g(n) = \Theta(1)$

   (b) If $c = 1$ then the series turns into $\sum_{i=1}^{n} 1$, which is just n, $\frac{n}{n} = 1$, so $g(n) = \Theta(n)$

   (c) If $c > 1$ then we can determine bounds by doing $\lim_{n\to\infty} \frac{g(n)}{c^n}$:

$$\lim_{n\to\infty} \frac{1 + c + c^2 + \; + c^n}{c^n}$$
$$\lim_{n\to\infty} \frac{1}{c^n} + \frac{c}{c^n} + ... + \frac{c^{n-1}}{c^n} + \frac{c^n}{c^n}$$
$$\lim_{n\to\infty} c^{-n} + c^{-n+1} + c^{-n+2} + \; + c^{-1} + 1 = \frac{1}{1 - \frac{1}{c}}$$

   This is a constant so we know $g(n) = \Theta(c^n)$

3. (a) $\begin{pmatrix} a & b \\ c & d \end{pmatrix} * \begin{pmatrix} e & f \\ g & e \end{pmatrix} = \begin{pmatrix} ae+bg & af+be \\ ce+dg & fc+ed \end{pmatrix}$ The result includes 8 multiplications, and then 4 additions

   (b) For any matrix X, we can compute X^n using $O(\log(n))$ matrix multiplications. We can do this using repeated squaring. For any n, we write the binary representation of the number. Let n_i represent the bit at position i (position 0 is the 1 bit, position 1 is the 2 bit, and so on). First, we find all the squares that we need. We calculate the following:
   X, X^2, $(X^2)^2 = X^4$, $(X^4)^2 = X^8$, ..., $X^{2^{\lceil \log_2(n) \rceil}}$
   Now for each n_i that is 1, we multiply together X^{2^i}
   The first step, there is one multiplication for each term except the first and there are $\log(n)$ terms, so that is $O(\log(n))$ and for the second half, there cannot be more than $\log(n)$ bits in n that are 1 because there cannot be more than $\log(n)$ bits in n. Therefore, there cannot be more than $\log(n)$ multiplications in this step. So the algorithm is $O(\log(n))$

   (c) Each entry in the resulting matrix after a matrix multiplication is composed of two multiplications and an addition (see (a)), all among different numbers. To begin with, at X^1, all of these numbers are 1 bit long, either 1 or 0. After one squaring, each entry in the matrix cannot be larger than 2 bits because a multiplication between two 1 bit numbers is at most 1 bit and addition of two 1 bit numbers can be two bits (for X^2). Now we have a matrix where each term can be at most 2 bits. We square that, each multiplication results in at most 4 bit numbers for X^4, the additions could make that 5 bits but we can safely ignore this possible additional bit because as we keep squaring and $n \to \infty$, the possible additional bit due to addition becomes irrelevant. Another square results in multiplying our 4 bit numbers together, getting 8 bit numbers as entries for X^8. Another square, and we get 16 bit entries for X^{16}. In general, because we started out with 1 bit numbers and because we are multiplying two of the entries each time, we effectively double the number of bits in each entry after each square. Therefore, the number of bits in an entry will be proportional to whatever exponent we raised X to, which cannot be larger than n, the final exponent of X. Therefore, we can say that the bit length of these entries are $O(n)$

   (d) We have to do $O(\log(n))$ squarings. Each squaring involves 8 multiplications and 4 additions of $O(n)$ bit numbers. Each squaring then takes $8 * M(n) + 4 * n = O(M(n))$ because $M(n)$ should dominate $4n$ as $n \to \infty$. So we have to do $O(\log(n))$ squarings, each taking $O(M(n))$ time, which is $O(M(n)\log(n))$. Once we have the squares, we have to multiply them together – these are again $O(M(n))$ and there are $O(\log(n))$ of them, so the total running time is $O(2M(n)\log(n)) = O(M(n)\log(n))$

(e) I found it confusing that n was used to be the exponent and its also the letter used for $M(n)$ for an n-bit number, just something I would look at/change for next semester.

We were generous with the multiplications last time, assuming matrix had entries of n bits, but really the number of bits increases exponentially, as shown in (c), so in theory only the later squarings with the really big numbers should count
We have $X, X^2, X^4, X^8, ..., X^{2^{\lceil \log_2(n) \rceil}}$
Which in terms of how long it takes to compute each term is
$O(M(1)) + O(M(2)) + O(M(4)) + O(M(8)) + ... + O(M(2^{\lceil \log_2(n) \rceil}))$
Because by (c) we know that each squaring has entries of bit length equal to the exponent. This whole sequence is dominated by the last term (by Q 2, part c, its the same idea), which is $O(M(n))$.

Now that we have computed the squares in $O(M(n))$ time, we have to multiply them together to get the total running time. The total time to multiply these squares cannot possibly take longer than $O(M(n))$.
$X * X^2 * X^4 * X^8 * ... * X^{2^{\lceil \log_2(n) \rceil - 1}} * X^{2^{\lceil \log_2(n) \rceil}}$
$O(M(2)) + O(M(4)) + O(M(8)) + ... + O(M(2^{\lfloor \log_2(n) \rfloor} - 1)) + O(M(2^{\lfloor \log_2(n) \rfloor}))$
Just like the summing the bits of a binary number, everything below the i^{th} bit cannot sum to higher than the i^{th} bit. The longest multiplication there could be would be two matrices of n bit numbers in the last multiplication on the right (assuming we order them correctly). Because this step takes $O(M(n))$, and the next highest multiplication is $(O(M(\frac{n}{2}))$, then $(O(M(\frac{n}{4}))$, etc the running time is dominated by the last multiplication and the entire algorithm is $O(M(n))$

4. We want to compute x^y. Let n be the number of bits in x, m the number of bits in y. The iterative algorithm does $y-1$ multiplications: $x*x*x*x*...*x$ To compute this, you do $x*x$, then x^2*x, until finally you have to compute $x^{y-1}*x$. The running times for all these is $O(n^2)+O(2n^2)+O(3n^2)+...+O((y-1)n^2)$ because there are n bits in each of the multiplications for the first term, then $2n$ bits and n bits, then $3n$ bits and n bits, and so on until we have $(y-1)n$ bits and n bits, which is $O((y-1)n^2)$. The total running time, therefore, of this way of squaring is $O(y^2n^2)$.

   Now lets try repeated squaring:
   $x, x^2, x^4, x^8, ..., x^{2^{m-1}}, x^{2^m}$
   $O(n^2), O(4n^2), O(16n^2), ..., O(2^{2m-2}n^2)$
   $O(n^2), O(4n^2), O(16n^2), ..., O(\frac{y^2}{4q}n^2)$
   This sequence is dominated by the last term, so the upper bound is $O(y^2n^2)$ Multiplying the squares together is at most $O(\log(y))$ multiplications of numbers no more than 2^mn bits long, and since $2^m \leq y$, the bit length is $O(yn)$. The last multiplication dominates (the one between two numbers of yn bits). The cost to multiply these numbers is $O(y^2n^2)$. So the repeated squaring algorithm takes $O(y^2n^2)$ time. Compared to the iterative approach, its about the same because the last multiplication in repeated squares is so expensive.

5. (a) We have n numbers to add, each of m bits. We have a circuit capable of adding these m bit numbers in $O(\log(m))$ circuit depth. In order to add all of these numbers in $O(\log(m)\log(n))$, we use an approach similar to repeated squaring. We take each pair of numbers (the first and second, third and fourth, fifth and sixth, etc) and add each in parallel. This reduces the total number of numbers we have to add from n to $\frac{n}{2}$. We do the same thing again, each time having the number of numbers we have to add. This takes $\log(n)$ iterations, each iteration taking approximately $O(\log(m))$ time, so the total running time it $O(\log(n)\log(m))$

   (b) We have three m bit numbers x, y, z. We want to be able to reduce this problem to adding r and s such that each bit of r and s can be computer independently of everything else. We can do this by using r_i to represent the parity of the addition of the i^{th} bits of x, y, and z (is the result odd, then r_i is 1) and s_{i+1} to represent whether the addition of the i^{th} bits of the former numbers produce a carry (or you can do s_i and then left shift s afterwards). That is, if the i^{th} digits are 0, 0, 0, then r_i and s_{i+1} are 0. If they are 1, 0, 0 (order unimportant), then $r_i = 1$, $s_{i+1} = 0$, if they are 1, 1, 0 (order unimportant), then $r_1 = 0$, $s_{i+1} = 1$, if they are all 1, then $r_i = 1$, $s_{i+1} = 1$. And $s_0 = 0$. Every bit of r and s can be computed independently of every other bit.

   (c) In binary, multiplying two n bit numbers is basically the same task as adding n n bit numbers. We can do this efficiently by reducing the computation of $a + b + c + ... + n$ to one using $\log(n)$ numbers, just as we did in part (b). However, this time we could potentially have more carry bits, and so we need to keep track of all of these (why we need $\log(n)$ numbers). First, find the sum of the i^{th} bits in the numbers we're adding. We will be able to express this sum in $\log(n)$ bits. Each of these will be the i^{th} bit of the resulting $\log(n)$ numbers, starting with the lowest bit (2^0) corresponding to the first number, 2^1 bit corresponding to the second bit and continuing from there. Do this for each of the bits in the sum. Now we have constructed $\log(n)$ numbers, the first having the least significant bit of the sums, the second having the next least significant bit of the sums, and so on. Now shift each left each number according to which bit it corresponds to (the power in the 2^x) – the first number has no shift, the next number corresponding to the 2^1 bit position gets shifted one, 2^2 gets shifted by 2, so on. Continue doing this until left with two numbers and then add them using the normal algorithm. This only requires a circuit of depth $\log(n)$