Advanced Algorithms, Fall 2011

Homework Set #1: Solutions

Question 1. Using any of your favorite solution methods, solve the following recurrences in Θ terms.

1. $f(n) = f(n/2) + 2f(n/4) + \Theta(n)$
2. $f(n) = f(n-1) \cdot f(n-2)$, with $f(1) = 1$ and $f(2) = 2$
3. $f(n) = \sum_{i=1}^{n-1} f(i)$ with $f(1) = 1$
4. a bit harder: let $f(n) = f(n-1)/2 + 2/f(n-1)$, with $f(0) = 3$, and define $g(n) = \Pi_{i=1}^{n} f(i)$ (hint: guess what the answer may be and verify it using limits of ratios)

1. $f(n) = f(n/2) + 2f(n/4) + \Theta(n)$
   Set $n = 2^k$ and $g(k) = f(2^k)$ to get $g(k) = g(k-1) + 2g(k-2) + \Theta(2^k)$. The characteristic equation is $x^2 - x - 2 = 0$, with two roots, $x_1 = 2$ and $x_2 = -1$. Thus the homogeneous solution is of the form $g_h(k) = a2^k = b(-1)^k$, which is dominated by the 2^k term. The driving function is also a 2^k term, so a particular solution is of the form $ck \cdot 2^k$ (the degree of the polynomial increases by one, as the radix is also a characteristic root). Thus, asymptotically, we have $g(k) = \Theta(k \cdot 2^k)$ and thus also $f(n) = \Theta(n \log n)$.

2. $f(n) = f(n-1) \cdot f(n-2)$, with $f(1) = 1$ and $f(2) = 2$ Perhaps the easiest way is to get rid of the multiplication by taking logs on both sides and setting $g(n) = \log f(n)$, to get $g(n) = g(n-1) + g(n-2)$, with $g(1) = 0$ and $g(2) = 1$ (assuming we are taking logarithms base 2), which is just our old friend the Fibonacci recurrence, but shifted by one; that is, we have $g(n) = F(n-1)$. Hence we have $f(n) = 2^{F(n-1)}$.

3. $f(n) = \sum_{i=1}^{n-1} f(i)$ with $f(1) = 1$ Let us transform this into a finite-order recurrence: $f(n) = \sum_{i=1}^{n-1} f(i) = \sum_{i=1}^{n-2} f(i) + f(n-1) = f(n-1) + f(n-1) = 2f(n-1)$ Now this is trivial: $f(n)$ is just 2^{n-2} for $n \geq 2$.

4. set $f(n) = f(n-1)/2 + 2/f(n-1)$, with $f(0) = 3$, and define $g(n) = \Pi_{i=1}^{n} f(i)$. First note that $f(n)$ is strictly decreasing and converges to 2 (both easy induction proofs). Now consider $h(n) = \log_2 g(n) = \sum_{i=1}^{n} \log_2 f(i)$; we have $\lim_{n\to\infty} \frac{h(n)}{n} = \log_2 2 = 1$. Thus $h(n)$ is $\Theta(n)$ and thus also $g(n)$ is $\Theta(2^n)$.

Question 2. In mathematics, one can write all sorts of complicated things with very simple notation; occasionally, some of these complicated things may even be expressible in a simpler way. So solve the following very complicated recurrence, which has a *very* simple solution:

$$f(x^2 + 4f(x) - 1) = f^2(x + 1)$$

Note that, if x^2 is asymptotically larger than $f(x)$, then this equation becomes, asymptotically, $f(x^2) = f^2(x)$, which is trivially satisfied by $f(x) = x$. The slight difference left is easily accounted by setting $f(x) = x + b$ and solving, to get $f(x) = x + 1$. Obviously, we could also have gotten there by substituting values.

Question 3. You are given an $N \times N$ matrix of distinct integers, sorted in increasing order along every row and along every column. You are also given a particular integer x that is present in the matrix. Devise and analyze an algorithm that, for any such matrix and any such integer, locates

the row and column where this element is to be found. You should base your algorithm on a binary search idea: get the "middle" element (initially, the element at row and column $N/2$), compare it with x, then search recursively in the appropriate three of the four $N/2 \times N/2$ submatrices around that midpoint.

The "appropriate" three of the four submatrices are all except the one matrix where every element eliminated by the query: if x is larger than the compared element, then the eliminated submatrix is the top left-hand one, whereas, if x is smaller than the compared element, the eliminated submatrix is the bottom right-hand one.

The algorithm uses a divide-and-conquer approach: the matrix is divided into four parts and recursion follows on three of the four parts, so the recurrence is $f(n) = 3f(n/4) + \Theta(1)$, where $n = N^2$ is the total number of elements in the matrix. As always in such cases, set $n = 4^k$ and write $g(k) = f(4^k)$ to get $g(k) = 3g(k-1)+\Theta(1)$, yielding $g(k) = \Theta(3^k)$ and thus $f(n) = \Theta(n^{\log_4 3})$. Thus the running time is sublinear, but not as spectacular as binary search, because of the need to explore 3/4, rather than 1/2, of the matrix in the recursive steps.

Question 4. Define the following approach for the multiplication for length-n binary vectors.

If we have $A = A[1 \ldots n] = A_L A_R$ and $A_L = A[1 \ldots n/2]$, $A_R = A[n/2 + 1 \ldots n]$, then we can write $XY = 2^n X_L Y_L + 2^{n/2}(X_L Y_R + X_R Y_L) + X_R Y_R$.

Now use this divide-and-conquer approach recursively until each vector has one element. Analyze the number of multiplications in this approach. Can you improve this approach? (Hint: $ab + cd = (a + c)(b + d) - ad - bc$)

The 2^n factor is due to the shift by $n/2$ bits of each of the two operands: the number represented by X is really $2^{n/2}X_L + X_R$ and so the product XY is really $(2^{n/2}X_L + X_R) \cdot (2^{n/2}Y_L + Y_R)$, which yields the expression given above. So what is the number of multiplications? This is something of a trick question, because the 2^n factor is just a shift and need not require a multiplication, while multiplying two length-one binary vectors is simply a logical AND and not a multiplication either Thus the number of multiplications in this approach is exactly zero!

Question 5. Matrix multiplication is easy to break into subproblems. For example, if we are given $A = \begin{vmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{vmatrix}$ and $B = \begin{vmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{vmatrix}$
then we can compute the product AB as follows:
$AB = \begin{vmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{vmatrix}$
(A and B are $(n \times n)$ matrices, and A_{ij} and $B_{i,j}$ are $(n/2 \times n/2)$ matrices.) Now use this divide-and-conquer approach recursively until each block has one element. Analyze the resulting number of multiplications.

Let $M(n)$ be the number of scalar multiplications used in multiplying two matrices, each of size $n \times n$. The recurrence set up in the problem breaks down the multiplication of A by B into 8 multiplication of matrices, each of size $n/2 \times n/2$. Thus we can write $M(n) = 8M(n/2)$, or, since n must be a power of 2, $n = 2^k$, $M(2^k) = 8M(2^{k-1})$. Solving as usual by writing $g(k) = M(2^k)$, we get $g(k) = 8^k$ and thus $M(n) = 8^{\log_2 n} = n^3$. Thus, interestingly, this rather naïve divide-and-conquer approach produces no gain at all—straightforward multiplication also uses n^3 multiplications. (It is possible to use the same divide-and-conquer scheme to run considerably faster, but this requires expressing the 8 subproducts in terms of a smaller number of common subproducts, so as to save on multiplications at the expense of extra additions and subtractions. This is the famous Strassen algorithm; in spirit, it is very close to the solution hinted at in the previous problem.)