CS 170: Problem Set 3

Due: September 16, 2011, 4:00 p.m.

Instructions: Turn this in to the class homework boxes in 283 Soda by 4:00 p.m. on Friday, September 16, 2011. Please begin your answer to each question on a new sheet of paper and make sure each sheet is labeled with your name, SID, section number, GSI name, the assignment number, the question number, and "CS 170 – Fall 2011."

Because each problem will be graded by a different reader, please **turn in each question in a different box in 283 Soda.** Question i goes in the box labeled "CS 170 — i."

Please read the class webpage for rules regarding collaboration (encouraged!) and cheating (forbidden!) on homework.

DPV = Dasgupta, Papadimitriou, and Vazirani.

1. DPV 2.7

2. DPV 2.9

   Note: In the textbook it says that to multiply two degree-d polynomials using the Fourier Transform with the nth roots of unity, you need $n \geq 2d + 1$. Something more general is actually true: To multiply a polynomial of degree d_1 by a polynomial of degree d_2, you just need $n \geq d_1 + d_2 + 1$. You may use this fact for this problem.

   Another note: For this problem, you must multiply the polynomials using the Fourier Transform, but you do not need to use the *Fast* Fourier Transform (FFT), even though the book says to use FFT.

3. DPV 2.30

4. (a) In the Master Theorem (as stated in the textbook and in lecture), we always assumed the base case is when the input size n is at most a constant, and the running time $T(n)$ is at most a constant. For this question, you are going to prove a more general version of the Master Theorem, where the base case is when the input size n is at most some parameter m, and the running time $T(n)$ is at most some parameter t. More precisely, consider a recurrence

$$T(n) \;\leq\; \begin{cases} t & \text{if } n \leq m \\ aT(n/b) + O(n^d) & \text{if } n > m \end{cases}$$

   Here, m and t are new parameters, and we want a big-O bound on $T(n)$ in terms of n, m, and t (note that a, b, and d are still constants). Prove that for all $n > m$,

$$T(n) \;\leq\; \begin{cases} O\big(n^d + t \cdot (\frac{n}{m})^{\log_b a}\big) & \text{if } d > \log_b a \\ O\big(n^d \log(\frac{n}{m}) + t \cdot (\frac{n}{m})^d\big) & \text{if } d = \log_b a \\ O\big((\frac{n}{m})^{\log_b a} \cdot (m^d + t)\big) & \text{if } d < \log_b a \end{cases}$$

   In the final answer, the constant factor hidden by the big-Oh notation is not allowed to depend on n, m, or t. (Note that if $m = \Theta(1)$ and $t = \Theta(1)$ then this coincides with the Master Theorem from the textbook and lecture.)

(b) In lecture we saw that the simplest divide-and-conquer algorithm for matrix multiplication (the one that reduces n-by-n matrix multiplication $C = A \cdot B$ to 8 multiplications and 4 additions of $\frac{n}{2}$-by-$\frac{n}{2}$ matrices) only moves $O(n^3/M^{1/2})$ words between main memory and a cache of size $M < n^2$. An analogous analysis can be done for Strassen's algorithm, showing that it only moves $O(n^w/M^x)$ words, where $w = \log_2 7 \approx 2.81$, and x is another constant. What is x? (Hint: Use part (a).)

5. We will show how to use a fast matrix multiplication algorithm (like Strassen's algorithm) to invert triangular matrices equally fast. A matrix T is upper triangular if all its entries below the main diagonal are zero, and also invertible if all its diagonal entries are nonzero. Assume T is n-by-n where for simplicity n is a power of 2.

(a) Let

$$T = \begin{bmatrix} T_{11} & T_{12} \\ 0 & T_{22} \end{bmatrix}$$

where each subblock T_{ij} is square with dimension $n/2$. Verify (by multiplying out $T \cdot T^{-1}$) that the inverse of T is given by

$$T^{-1} = \begin{bmatrix} T_{11}^{-1} & -T_{11}^{-1} \cdot T_{12} \cdot T_{22}^{-1} \\ 0 & T_{22}^{-1} \end{bmatrix}$$

(b) Assuming you have a function $C = matmul(A, B)$ for computing the product of two matrices $C = A \cdot B$, use it with the above formula to design a divide-and-conquer algorithm $triinv(T)$ for inverting upper triangular matrices. (Look at the class webpage for instructions on how format your answer to a question that asks you to design an algorithm. The first three steps of this 5-step guideline are covered by this part, while the last two steps are covered by part (c) below.)

(c) Suppose that the cost of $matmul(A, B)$ is $O(n^w)$ when run on n-by-n input matrices. For example, for the conventional algorithm $w = 3$ and for Strassen $w = \log_2 7 \approx 2.81$. Assuming $2 \le w \le 3$, show that $triinv(T)$ also costs $O(n^w)$ on n-by-n input matrices (that are upper triangular and invertible). Why can't w be less than 2?

(d) Extra credit: Suppose you have a parallel computer with p processors, and a parallel matrix multiplication function $C = pmatmul(A, B)$ that runs in $O(n^w/p)$ steps. Write down a parallel divide-and-conquer version $ptriinv(T)$ of your earlier algorithm.

(e) Extra credit: What is the complexity of $ptriinv(T)$, as a function of n and p? Assume that at each recursive call, half the available processors work on T_{11} and half work on T_{22}. Hint: For simplicity you can assume n and p are powers of 2, and $p \ge n$.

(f) Extra credit: Do the case $p < n$ as well (but still a power of 2).

Comment (you don't need to do this): The above techniques can be generalized to the inversion of a general (full, not triangular) matrix, computing determinants, and similar problems. These depend on the formula (which can be thought of as one step of "block" Gaussian elimination):

$$A = \begin{bmatrix} B & C \\ D & E \end{bmatrix} = \begin{bmatrix} I & 0 \\ D \cdot B^{-1} & I \end{bmatrix} \cdot \begin{bmatrix} B & C \\ 0 & E - D \cdot B^{-1} \cdot C \end{bmatrix} .$$

This divide-and-conquer formula implies both that (here $S = E - D \cdot B^{-1} \cdot C$)

$$\begin{aligned} A^{-1} &= \left(\begin{bmatrix} I & 0 \\ D \cdot B^{-1} & I \end{bmatrix} \cdot \begin{bmatrix} B & C \\ 0 & S \end{bmatrix} \right)^{-1} \\ &= \begin{bmatrix} B & C \\ 0 & S \end{bmatrix}^{-1} \cdot \begin{bmatrix} I & 0 \\ D \cdot B^{-1} & I \end{bmatrix}^{-1} \\ &= \begin{bmatrix} B^{-1} & -B^{-1} \cdot C \cdot S^{-1} \\ 0 & S^{-1} \end{bmatrix} \cdot \begin{bmatrix} I & 0 \\ -D \cdot B^{-1} & I \end{bmatrix} \\ &= \begin{bmatrix} B^{-1} + B^{-1} \cdot C \cdot S^{-1} \cdot D \cdot B^{-1} & -B^{-1} \cdot C \cdot S^{-1} \\ -S^{-1} \cdot D \cdot B^{-1} & S^{-1} \end{bmatrix} \end{aligned}$$

and

$$det(A) = det(B) \cdot det(S)$$

which reduce the n-by-n inversion and determinant problems to two $n/2$-by-$n/2$ problems.

You may object that the algorithm has a bug: it could fail if some submatrix being inverted (like B) is singular, even if the full matrix is not. There are ways to avoid this.