

## CS303 (Spring 2008) — Assignment 7

Due: 03/05/2008

- (1) Implement (in C, C++, or Java) the two algorithms for integer multiplication. One of the algorithms should be the  $\Theta(n^2)$  one you learned in elementary school, the other one the  $\Theta(n^{1.59})$  Divide&Conquer algorithm from class. The input and output to your program should be two numbers represented as arrays of integers/bits. That is, you should not represent numbers as `integer` or `long` or something like that (that would restrict you to 32 or 64 bits, and for that many bits, you won't need any sophisticated algorithms). You are welcome to implement your algorithm base 2 or base 10, according to your choice. In order to make this work, you'll probably also need to implement addition/subtraction for numbers stored as arrays.

By using the timing code you wrote for a previous homework, find out for how many digits  $n$  the  $\Theta(n^2)$  algorithm beats the  $\Theta(n^{1.59})$  and when the Divide&Conquer algorithm starts beating the elementary one.

- (2) Problem 28.2-4 from the textbook. Justify your answer, i.e., don't just say which one is best, but also why. (Hint: think about what exactly Strassen's algorithm does.)
- (3) Problem 28.2-6 from the textbook. Also briefly justify your answer here.
- (4) Imagine that you are given a data structure  $a$  with  $n$  elements in it. Your goal is to find out whether *more than* half (i.e., at least  $\lfloor n/2 \rfloor + 1$ ) of these elements in the data structure are all identical. However, you can only access the elements via a subroutine `areEqual(a, i, j)` which returns (in constant time  $\Theta(1)$ ) `true` if elements  $i$  and  $j$  in the data structure are equal, and `false` otherwise. In particular, you cannot find out the actual values of the elements, or sort them, or do anything like that.

Give and analyze an algorithm with running time  $O(n \log n)$  for finding out if more than half of the elements are the same. In particular, you can thus not call `areEqual` more than  $O(n \log n)$  times. Your algorithm need not actually *find* the identical elements; it's enough if it always correctly says "Yes" or "No".