CSE 421 Intro to Algorithms Autumn 2017

Homework 4

Due Friday October 27, 4:00 pm

Problem 1:

Given an array of elements $\mathbf{A}[1,\ldots,n]$, give an $O(n \log n)$ time algorithm to find a majority element, namely an element that is stored in more than $n/2$ locations, if one exists. Note that the elements of the array are not necessarily integers, so you can only check whether two elements are equal or not, and not whether one is larger than the other.

HINT: Observe that if there is a majority element in the whole array, then it must also be a majority element in either the first half of the array or the second half of the array.

Problem 2:

Show how to multiply two degree 2 polynomials using fewer multiplications of coefficients than the naive algorithm. You can do it with only 5 multiplications (not counting multiplications by constants). (For partial credit, show how to do this with only 6 multiplications.) Use this to describe an algorithm for polynomial multiplication that is more efficient than Karatsuba's algorithm.

Why wouldn't 6 multiplications be enough to be better than Karatsuba's algorithm?

HINT: Use evaluation and interpolation.

Problem 3:

Hidden surface removal is a problem in computer graphics that scarcely needs an introduction: when Woody is standing in front of Buzz, you should be able to see Woody but not Buzz; when Buzz is standing in front of Woody, . . . well, you get the idea.

The magic of hidden surface removal is that you can often compute things faster than your intuition suggests. Here's a clean geometric example to illustrate a basic speed-up that can be achieved. You are given n nonvertical lines in the plane, labeled $L_1, \ldots, L_n$, with the i^{th} line specified by the equation $y = a_i x + b_i$. We will make the assumption that no three of the lines all meet at a single point. We say line L_i is *uppermost* at a given x-coordinate x_0 if its y-coordinate at x_0 is greater than the y-coordinates of all the other lines at x_0: $a_i x_0 + b_i > a_j x_0 + b_j$ for all $j \neq i$. We say line L_i is *visible* if there is some x-coordinate at which it is uppermost—intuitively, some portion of it can be seen if you look down from "$y = \infty$."

Give an algorithm that takes n lines as input and in $O(n \log n)$ time returns all of the ones that are visible. Figure 5.10 gives an example.

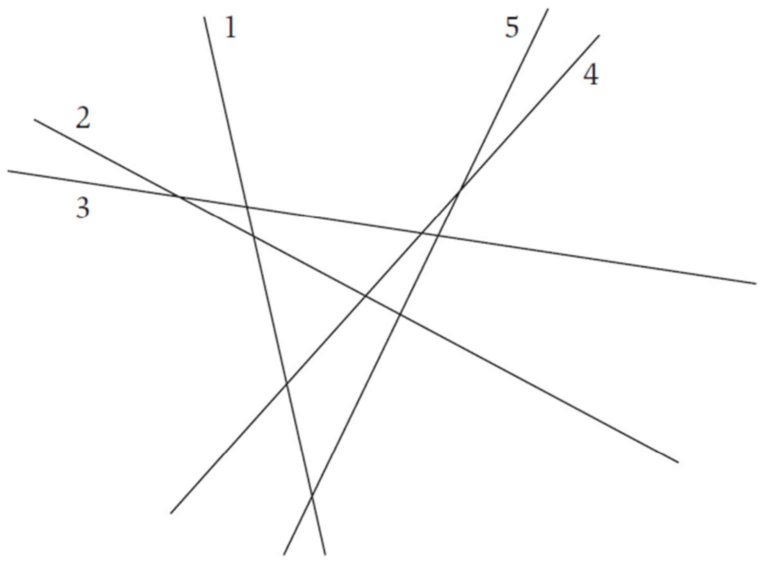

Figure 5.10 An instance of hidden surface removal with five lines (labeled *1–5* in the figure). All the lines except for *2* are visible.

Problem 4 (Extra Credit):

(Due Monday, November 6, 11:59pm - may be done with a partner - list your partner on your homework.)

In describing and analyzing Strassen's algorithm we assumed that we used divide and conquer all the way down to tiny matrices. However, on small matrices the ordinary matrix multiplication algorithm will be faster because of lower overhead. This is a common issue with divide and conquer algorithms. The best way to run these algorithms typically is to test the input size **n** at the start to see if it is big enough to make using divide and conquer worthwhile; if **n** is larger than some threshold **t** then the algorithm would do a level of recursion, if **n** is below that threshold then it would do the non-recursive algorithm.

Your job in this question is to figure out the best choice for that threshold value for a version of Strassen's algorithm *over the integers* based on your implementation. (See the class slides for the description of the recursion used in Strassen's algorithm and for the code for the basic non-recursive algorithm for matrix multiplication.)

You should code up the pure algorithms first and then create the final hybrid algorithm. For simplicity you can assume that the size **n** of the matrix is a power of 2 and figure out the matrix size $\mathbf{t=2^i}$ below which it is better to switch to the ordinary algorithm.

Your goal is to beat the ordinary algorithm by as much as possible and so find the smallest cross-over point you can. The language you choose to implement this in is somewhat up to you. However, the object-oriented implementation of two-dimensional arrays in Java with most of its standard class libraries is not great for working with two-dimensional sub-arrays. Using a language such as C that has more efficient array implementations and can use integer arithmetic on the array indices to let you identify submatrices without copying them will give you better results. Check your answer for correctness against the naive algorithm. For your solution upload a PDF of your code, information on your test inputs, the timings that you found, and the choice of **t** that you found works best.