CSE 421 Winter 2021
Homework 4

Due: Friday, February 5, 2021, 6:00 pm

Problem 1:
Given a 1-dimensional array of n real numbers, consider the problem of finding the following three items:

- the contiguous sub-array of the input with maximum sum.
- the prefix of the array with maximum sum, and
- the suffix of the array with maximum sum.

For example, in the array

$$\{31, -49, 59, 26, -53, 58, 97, -93, -23, 54\}$$

- the contiguous sub-array with maximum sum is achieved by summing the 3rd through 7th elements, whose sum is 187. (When all numbers are positive the answer would be the whole array; when all numbers are negative the answer wold be an empty sub-array which has a total of 0.)
- the prefix sum is maximized with the prefix ending in the 7th element, which has sum 169.
- the suffix sum is maximized with the suffix that begins with the 3rd element, which has sum 125.

Give an $O(n)$ *divide and conquer* algorithm to find all three of these segments in an array of n numbers and compute their associated sums.

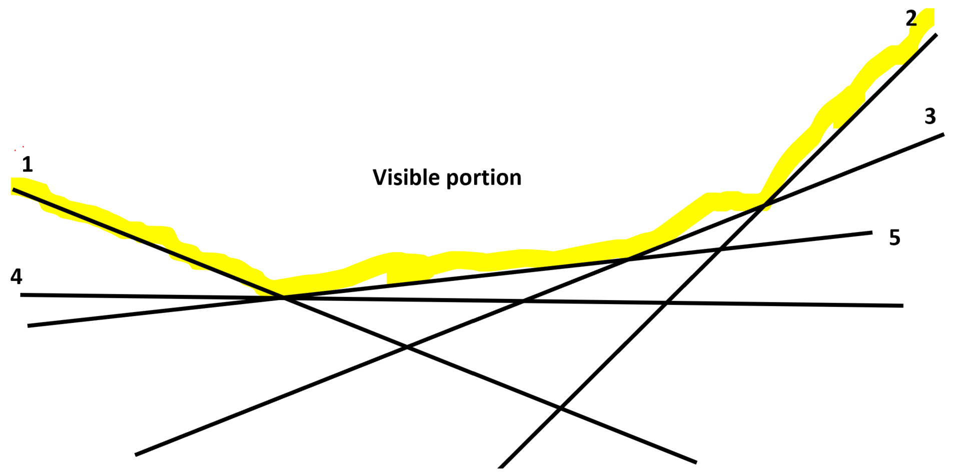
Problem 2:
In computer graphics, images are generally produced from geometric models of surfaces that consist of sets of triangles, each of which has a position and orientation in space. The resulting image is found by considering a positions seen from the point of view of a "camera" and the image is based on the portions of triangles the camera can see. (The visible portions of these triangles are later processed to handle the effects of light, but this first process can greatly simplify the portion of the models that is needed to produce the image.)

In this question we focus on a simplified 1-dimensional version of the problem. The natural 1-dimensional version corresponding to triangles would involve line segments, but we will simply consider the processing required for the case for a set L of infinite *lines* in the xy-plane of the form $y = a \cdot x + b$. (Note that this rules out vertical lines.) Furthermore, instead of a "camera" at single spot we assume that the visible portion is what would be seen by an "airplane" flying infinitely high over the terrain given by the lines. Visibility at a given x value therefore just means has the highest y value at x.

The resulting visible portion of the scene given by these lines will consist of a region that has half-lines at each end and line segments in the middle. The visible region can then be described by a sequence

$$V = (\ell_1, x_1, \ell_2, x_2, \ell_3, x_3, \ldots, \ell_{t-1}, x_{t_1}, \ell_t)$$

for some t where each $\ell_i \in L$, each $x_i \in \mathbb{R}$ and $x_1 < x_2 < \ldots < x_{t-1}$. The meaning is that line ℓ_1 is visible from $x = -\infty$ until $x = x_1$ at which point line ℓ_2 becomes visible until $x = x_2$, etc. (We assume no duplicate lines in L and among all lines that might agree on a highest y value at a given point x_j, we only include the two lines that are highest immediately to the left and right of $x = x_j$. For example, in the scene below, the lines in the visible portion in sequence would be lines 1, 5, 3, and 2 in order. (Without a coordinate system we can't define the x_i values in V.)

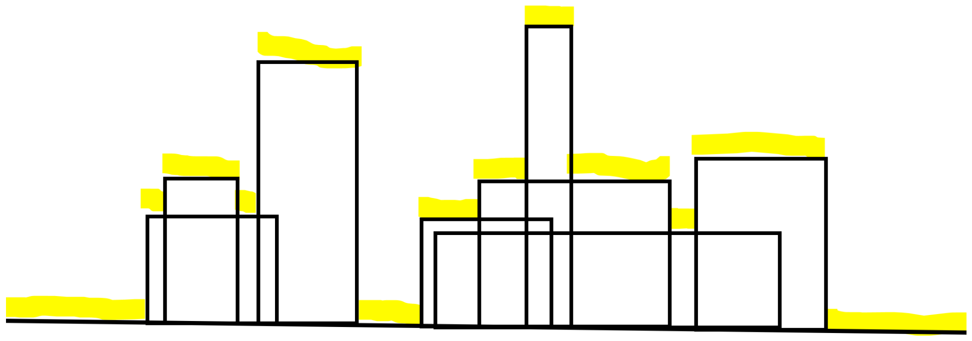


Given an $O(n \log n)$ time algorithm (assuming any of the usual arithmetic operations on real numbers can be done in one time step) that takes a set L of n lines expressed as a sequence of pairs (a_i, b_i) of real numbers $y = a_i \cdot x + b_i$ produces the sequence V defining the scene.

Problem 3:
This problem is another one-dimensional variant of Problem 2 where instead of infinite lines, the objects in the scene are boxes sitting on the x-axis (a line at height 0). Each box (rectangle) is given by a triple (x_L, h, x_R) where $x_L < x_R$ and $h > 0$ indicating the point where it starts, its height, and where it ends. Your goal is similar to what it was in Problem 2 of describing the visible (highest) surface at all points, except now we represent what is visible by a sequence

$$S = (x_1, h_1, x_2, h_3, x_3, \cdots, h_{t-1}, x_t)$$

where x_1 and x_t respectively are the smallest and largest points at which the height is > 0 and h_i is the height of the highest points between x_i and x_{i+1}. If there is no box at a point then the height will be 0. (See the associated diagram.)



Give an $O(n \log n)$ algorithm to find the visibility sequence S for an input set of n boxes as above.

Problem 4 (Extra Credit):
Use the same ideas as used to solve the problem for closest pair in the plane to create an $O(n \log^2 n)$ algorithm for finding closest pairs in 3 dimensions and prove its running time. (This is not optimal; an $O(n \log n)$ algorithm exists that uses ideas of the above algorithm plus more flexibility in choosing how to split the points into subproblems so that the difficult strip in the middle has fewer points.)

Problem 5 (Extra Credit): Due February 15.
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 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 $t = 2^i$ 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 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 2-dimensional array implementations that can use integer arithmetic on the array indices to let you iterate through 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.