I500/B609: Fundamental Computer Concepts of Informatics
Discussion problems (week 2, divide-and-conquer algorithms)

1. Given a sorted array of distinct integers A[1..n], you want to find out if there is an index i for which A[i] = i. Devise a divide-and-conquer algorithm that runs in time O(log n).
2. (k-way merge) Suppose you are given *k* sorted arrays, each with n elements, and you want to combine them into a single sorted array of *kn* elements.
    a. Here is one strategy. Using the merge procedure to merge the first two arrays, then merge in the third, and so on. What is the time complexity of this algorithm, in terms of *k* and *n*?
    b. Devise a more efficient algorithm using the divide-and-conquer approach.
3. An array A[1…n] is said to have a *majority element* if more than half of its entries are the same. Given an array, the task is to design an efficient algorithm to tell if the array has a majority element, and if so, to find the element. The elements of the array are not necessarily from some ordered domain like the integers, and so there can be no comparisons of the form "is A[i] > A[j]?". However, you can answer the questions of the form: "is A[i] = A[j]?" in constant time.
    a. Show how to solve this problem in O(n log n) time.
    b. Can you devise a linear time algorithm for this problem?

4. Consider the following *closest pair* problem.
   Input: A set of points in the plane, $\{p_1 = (x_1, y_1), \ldots, p_n = (x_n, y_n)\}$
   Output: the closest pair of points, that is the pair $p_i \neq p_j$, for which the distant between p_i and p_j is minimized. For simplicity, assume n is the power of two, and that all the x-coordinates and y-coordinates of the points are distinct. Here is a sketch of a divide-and-conquer algorithm to solve the problem.
    a. Find a value x for which exactly half of the points have $x_i < x$ and half have $x_i > x$; split the set of points into two groups: L and R.
    b. Recursively find the closest pair in L and in R. Let these pairs are $(p_L\ q_L)$ and (p_R, q_R) with distances of d_L and d_R, respectively. Let d be the smaller of these two distances.
    c. It remains to be seen whether there is a point in L and a point in R that are less than the distance d apart from each other. To this end, discard all points with $x_i < x-d$, and $x_i > x+d$, and sort the remaining points by their y coordinates.
    d. Now go through the sorted list, and for each point, compute its distance to the *seven* subsequent points in the list. Let p_M and q_M be the closest pair found in this way.
    e. The answer is one of the three pairs $(p_L\ q_L)$, $(p_R\ q_R)$ and $(p_M\ q_M)$.
    (1) In order to prove the correctness of the algorithm, start by showing the following property: any square of size d×d in the plane contains at most four points of L.
    (2) Now show that the algorithm is correct.

(3) Show the running time of the algorithm is given by the recurrence: $T(n) = 2T(n/2) + O(n \log n)$, and thus the run time of the algorithm is $O(n \log^2 n)$

5. Let F_n be the *n*th Fibonacci number. F_n can be computed in the matrix form:

$$\begin{pmatrix} F_1 \\ F_2 \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \cdot \begin{pmatrix} F_0 \\ F_1 \end{pmatrix}$$

$$\begin{pmatrix} F_2 \\ F_3 \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \cdot \begin{pmatrix} F_1 \\ F_2 \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^2 \cdot \begin{pmatrix} F_0 \\ F_1 \end{pmatrix}$$

… …

$$\begin{pmatrix} F_n \\ F_{n+1} \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^n \cdot \begin{pmatrix} F_0 \\ F_1 \end{pmatrix}$$

So, in order to compute F_n, it suffices to rate the 2×2 matrix (denoted as X) to the *n*th power.

(1) Show that the two 2×2 matrices can be multiplied using four additions and 8 multiplications;
(2) Devise an algorithm to compute X^n that takes O(log n) times of matrix multiplication;

It seems we have an algorithm that needs only O(log n) time to compute the Fibonacci number. However, the catch is that the algorithm involves multiplication, not just addition; and the multiplication of large numbers is slower than addition.

(3) Let M(n) be the running time of an algorithm of multiplying n-bit numbers. Show that the run time of the new algorithm is O(M(n) log n).
(4) Can you show the run time of the algorithm is O(M(n)), if $M(n)=O(n^a)$ for some $1<a\leq 2$ (Hint: the bit-length of the numbers being multiplied get doubled with every squaring).