Divide and conquer

For *all* of the following problems, a proof that the algorithm you design is correct is required.

1. For a sequence of n numbers $a_1, \ldots, a_n$, which we assume are all distinct, we define a *significant inversion* to be a pair (a_i, a_j) such that if $i < j$ and $a_i > 2a_j$. Give an $O(n \log n)$ algorithm to count the number of significant inversions between two orderings.

2. Let MERGE(A, B) be the linear-time procedure that takes two sorted arrays, A and B, and returns a single sorted array that contains the elements $A \cup B$.

   Suppose you have k sorted arrays, each with n elements, and you want to combine them into a single sorted array.

   (a) What is the time complexity of the algorithm that merges the first two arrays, then merges in the third array, then the fourth, and so on?

   (b) Give a more efficient solution to this problem. Give pseudocode (using MERGE(A, B) as a black-box) and analyze the running time.

   Note that your running times should depend on both k and n.

3. You are given two sorted lists of size m and n. Give an $O(\log m + \log n)$ time algorithm for computing the k^{th} smallest element in the union of the two lists.

4. Suppose we are given an array $A[1 \ldots n]$ with the special property that $A[1] \geq A[2]$ and $A[n-1] \leq A[n]$. We say that an element $A[x]$ is a local minimum if it is less than or equal to both its neighbors, or more formally, if $A[x1] \geq A[x]$ and $A[x] \leq A[x+1]$. For example, there are six local minima in the following array:

   $$[9\ 7\ 7\ 2\ 1\ 3\ 7\ 5\ 4\ 7\ 3\ 3\ 4\ 8\ 6\ 9]$$

   We can obviously find a local minimum in $O(n)$ time by scanning through the array. Describe and analyze an algorithm that finds a local minimum in $O(\log n)$ time.

5. *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 of three of the lines 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 y-coordinate of all the other lines at x_0 : $a_i x_0 + b_i > a_j x_0 + b_j \quad \forall j \neq i$. 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.

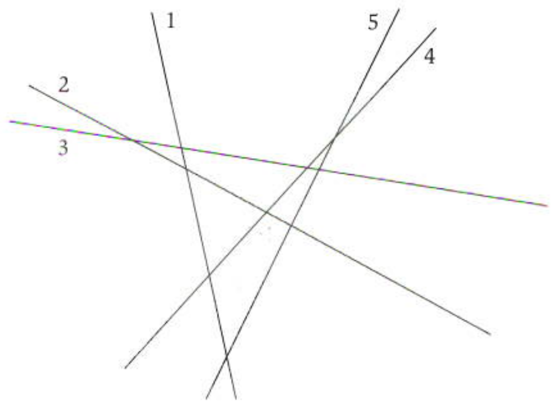

Figure 5.10 An instance of hidden surface removal with five lines (labeled *1–5* in the figure). All the lines except for *2* are visible.

6. You are at a political convention with n delegates. Each delegate is a member of exactly one political party. It is impossible to tell which political party a delegate belongs to. However, you can check whether any two delegates are in the *same* party or not by introducing them to each other. (Members of the same party always greet each other with smiles and friendly handshakes; members of different parties always greet each other with angry stares and insults.)

   (a) Suppose a majority (more than half) of the delegates are from the same political party. Give an efficient algorithm that identifies a member of the majority party. Hint: perhaps the usual divide approach is not the best one.

   (b) Suppose exactly k political parties are represented at the convention and one party has a *plurality*: more delegates belong to that party than to any other. Give an efficient algorithm that identifies a member of the plurality party.