Question 1. [Before working on this question, review Section 4.6: Union-Find]

This question explores a data structure that supports a specific set of operations on an string of \( n \) bits. Initially, all bits are set to 0. Bits can then individually be set to 1 (but not back to 0), and the value of the \( i^{th} \) bit can be obtained. As more bits are set to 1, blocks of consecutive 1s may be formed, separated by 0s (or terminating at the ends of the string). Specifically, the BitBlocks data structure supports the following methods:

- **Create\( (n) \):** Initialize the data structure.
- **SetToOne\( (i) \):** Given an index \( i \), set \( i^{th} \) bit to 1 (even if it already is 1).
- **GetValue\( (i) \):** Given an index \( i \), return the value of the \( i^{th} \) bit.
- **GetBlockSize\( (i) \):** If GetValue\( (i) = 1 \), return the size of the block containing the \( i^{th} \) bit; otherwise return 0.

We would like to ensure that any sequence of these operations, beginning with the initializing of the data structure, will be highly efficient. In particular, we’ll allow Create to take \( O(n) \) time, but require any subsequent sequence of SetToOne, GetValue, and GetBlockSize operations to be very fast. The data structure should be of size no worse than \( O(n) \).

(a) Describe an implementation of the BitBlocks data structure based on a Union-Find data structure. Include descriptions of the implementations of the four methods defined above. Hint: In addition to the U-F structure, maintain an array of \( n \) bits and another array that contains some kind of helpful size information.

(b) Prove that the methods of your data structure work as specified above.

(c) Analyze the time and space complexity of your data structure and each of its methods. In particular, state and prove a result analogous to statement (4.23) [page 153] of your text.

Question 2. Suppose you are choosing between the following three algorithms:

1. Algorithm A solves problems by dividing them into five subproblems of half the size, recursively solving each subproblem, and then combining the solutions in linear time.
2. Algorithm B solves problems of size \( n \) by recursively solving two subproblems of size \( n - 1 \) and then combining the solutions in constant time.
3. Algorithm C solves problems of size \( n \) by dividing them into nine subproblems of size \( n/3 \), recursively solving each subproblem, and then combining the solutions in \( O(n^2) \) time.

What are the running times of each of these algorithms (in asymptotic notation) and which would you choose?

Question 3. An array \( A[1 \ldots n] \) 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 whether the array has a majority element, and, if so, to find that 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 \( A[i] > A[j] \). You should think of the array elements as, say, JPEG files. However, you can answer questions of the form: \( A[i] = A[j] \) in \( O(1) \) time.

(a) Show how to solve this problem in \( O(n \log n) \) time. **Hint:** split the array \( A \) into two arrays \( A_1 \) and \( A_2 \) of half the size. Does knowing the majority elements of \( A_1 \) and \( A_2 \) (if such elements exist) help you figure out the majority element of \( A \)? If so, can you use a divide-and-conquer approach? Be sure to prove that your algorithm is correct (via induction) and give a recurrence for the running time of your algorithm.  

(b) Can you design a linear-time algorithm? **Hint:** arbitrarily pair up the elements of the array. For each pair, if the two elements are different, discard both of them; if they are the same, keep just one of them. Show that after this procedure executes, there are at most \( n/2 \) elements left, and that they have a majority element if \( A \) does.

Question 4. Solve Problem 5 from Chapter 5 of your text. Divide and conquer works well here but there are other methods of solution; feel free to try a different way if D-C isn’t working for you!

**Hint:** What does the structure of the set of all uppermost points for a set of lines \( L \) look like? If you had these structures for sets \( L_1 \) and \( L_2 \) of lines, could you compute the structure for \( L_1 \cup L_2 \)?

---

1 Your time complexity analysis can assume that \( n \) is, say, a power of 2, but your correctness argument must deal with all values of \( n \),
Question 5. Solve Problem 6 from Chapter 5 of your text. Clarification. What does probe mean? Imagine that for each vertex \( v \) in the tree, the label \( x_v \) is a quantity that must be computed and that is very time-consuming to compute \( x_v \). A probe of \( v \) is a computation of \( x_v \), so we want to minimize the number of such operations.

Optional Challenge Problem!

Note: These problems are intended to challenge/entertain/enlighten and provide opportunities for further practice with no added stress (think of them as the WSP of homework problems...).

Question 6. A convex polygon in the plane is one in which, for every two points \( x, y \) on the polygon, the line segment from \( x \) to \( y \) is contained in (or on the boundary of) the polygon. Convex polygons arise frequently. Given a set \( P \) of \( n \) points in the plane, no 3 of which are collinear, the convex hull of \( P \) is defined to be the smallest convex polygon containing \( P \). The corners of the convex hull of \( P \) are the "outermost" elements of \( P \).

[a] Design an algorithm to compute the convex hull of \( P \) that runs in \( O(n \log n) \) time. Justify its correctness and running time.

[b] Show that given the convex hull of \( P \) one can compute in linear time (actually in time proportional to the number of vertices on the convex hull) the diameter of \( P \)—that is, the furthest distance between any two points of \( P \).