Answer all questions. Give informal (at least) proofs for all answers. Grading will be on completeness and logical correctness, and if applicable, efficiency, as well as correctness.

**Analyzing loops-10pts** Consider the following algorithm, that given an array of integers $A[1..n]$, for each $1 \leq I \leq n$, finds the last position $1 \leq J \leq I$ with $A[J] > A[I]$ (or 0 if no such $J$ exists), and stores it in $B[I]$.

*(LastLargerElement[$A[1..n]$: array of integers]*)
1. FOR $I = 1$ to $n$ do:
2. 
5. Return $B$

Give a worst-case time analysis, up to $\Theta$, for this algorithm, as a function of $n$. (Since there are some inputs on which this algorithm is faster than its worst-case, be sure to provide an example of inputs on which its performance matches the analysis.)

For an upper bound, observe that the while loop runs at most $I$ iterations, each of which is constant time. Thus, since $I \leq n$, it is $O(n)$ time per while loop. The other lines of the for loop are constant time, so the total time of the nested loops is at most $O(n^2)$.

For a lower bound on the worst-case time complexity, consider the case when the input is sorted from smallest to largest. Then since $A[J] \leq A[I]$ for each $1 \leq J < I$, the inside while loop will decrement $J$ $I$ times, so is at least $\Omega(I)$ in time. For $I$ ranging between $n/2$ and $n$, this is $\Omega(n)$ per each $I$, for a total time that is $n/2 \cdot \Omega(n) = \Omega(n^2)$.

Thus the worst-case running time is $\Theta(n^2)$. (Note that this doesn’t mean that the running time is $\Theta(n^2)$ on every input; it means that it is at most the bound for EVERY input, and at least the bound for SOME inputs. But if the input is sorted the other way, the algorithm will be $O(n)$ time.)

**Correctness proofs** Consider the last larger element problem described above. Here’s a high-level algorithmic strategy for this problem:

LastLargerElement($A[1..n]$):
1. $B[1] \leftarrow 0$.
2. Initialize $S$ as $\{1\}$.
3. For $I = 2$ TO $n$ do:
4. 
5. While $J > 0$ and $A[J] \leq A[I]$ do:
6. 
7. IF $S \neq \emptyset$
8. 
9. ELSE $J \leftarrow 0$
11. Insert $I$ into $S$.
12. Return $B$

1. Prove that if \( J \) is the last larger element to \( A[I + 1] \), then \( J \in S_I \). (3 points)

Since \( J \) is the position of the last larger element before \( I + 1 \), \( A[J] > A[I + 1] \) but \( A[I + 1] \geq A[K] \) for all \( J < K \leq I \) (or else \( A[K] \) would be a later element larger than \( A[I + 1] \).) Thus, \( A[J] > A[K] \) for all such \( K \), and is hence unblocked at \( I \).

2. Prove that if \( 1 \leq J_1 < J_2 < \ldots J_k \leq I \) are the elements of \( S_I \), then \( A[J_1] > A[J_2] > \ldots A[J_k] \). (2 points)

Since \( J_i \) is unblocked, \( A[J_i] > A[K] \) for all \( J_i < K \leq I \). In particular, \( J_t < J_{t+1} \leq I \), so we can pick \( K = J_{t+1} \) to conclude \( A[J_t] > A[J_{t+1}] \).

3. Prove the following loop invariant: after the loop, \( S = S_I \). (3 points)

\( 1 \leq J \leq I \) is unblocked at \( I + 1 \) if and only if it is unblocked at \( I \) and \( A[J] > A[I + 1] \). Also, \( I + 1 \) is always unblocked at \( I + 1 \), since there are no \( I + 1 < K \leq I + 1 \). Thus, \( S_{I+1} = J \in S_I \mid A[J] > A[I + 1] \cup \{ I + 1 \} \). Assume that \( J \in S_I \) is the largest element of \( S_I \) with \( A[J] > A[I + 1] \). Then in the \( I \)th iteration, our algorithm deletes all elements of \( S = S_I \) that are larger than \( J \) and keeps all the elements that are smaller than \( J \). But all the elements larger than \( J \) have \( A[J] < A[I + 1] \) and hence are blocked. By part 2, for all elements \( J' \leq J \) of \( S_I \), \( A[J'] \geq A[J] > A[I + 1] \), so all such elements are in \( S_{I+1} \). Thus, \( S \) agrees with \( S_{I+1} \) on all elements at most \( I \). At the end, we insert \( I + 1 \), so \( S \) is \( S_{I+1} \).

4. Use this to conclude that each \( B[I] \) is the position of the last larger element before \( A[I] \) (or 0 if none exists). (2 points)

Assume there is a larger previous value before \( A[I] \), and let \( J \) be the last larger position. By part 1, \( J \in S_{I-1} \), which by the invariant is \( S \) before the \( I \)th loop begins. Any later value \( J \leq J' \leq I \) in \( S \) has \( A[J'] \leq A[I] \), so all such values are deleted in the loop. Since \( A[J] > A[I] \), \( J \) is not deleted, and \( B[I] \) is assigned \( J \), which is the last larger position. If there are no previous larger positions, then the loop can only end after all elements of \( S \) have been deleted, when we assign \( B[I] \) the value 0.

Data structures and efficient versions of algorithms 10 pts: For the last larger element problem and strategy above, give an efficient algorithm to find the last larger element based on the given strategy. Specify clearly the data structures and preprocessing used, and give pseudo-code or a clear description of all steps in terms of these data structure operations. Give an informal explanation for why your algorithm follows the given strategy. Give a complete time analysis of your algorithm. Some of your grade will be based on the efficiency of your algorithm.

Note that in the loop, we need to find and possibly delete the largest element of \( S \) and insert one element \((I + 1)\). Thus, a heap would seem to be a good match. Using a heap, the algorithm becomes:

LastLargerElement\((A[1..n])\):

1. \( B[I] \leftarrow 0. \)
2. Initialize a max-heap \( S \) as \( \{1\} \).
3. For \( I = 2 \) TO \( n \) do:
   4. \( J \leftarrow \text{FindMax}.S \)
   5. While \( J > 0 \) and \( A[J] \leq A[I] \) do:
      6. \( \text{DeleteMax}.S \)
      7. IF \( \text{FindMax}.S \neq \text{NIL} \)
         8. THEN \( J \leftarrow \text{FindMax}.S \)
      9. ELSE \( J \leftarrow 0 \)
   10. \( B[I] \leftarrow J. \)
11. \( \text{Insert}.S(I). \)
12. Return \( B \)
Note that each element is inserted once, in the last line of the for loop. Thus, there are \( n \) insertions total which means there are at most \( n \) deletions total. Since inserts and deletes in a heap each cost \( O(\log n) \), and the size of the heap is at most \( n \), this is a total of \( O(n \log n) \) time for all insertions and deletions. Since we delete at least once each iteration of the While loop, and we delete at most \( n \) times total, the while loop has a total of \( n \) iterations throughout the life of the algorithm. This means the total time for the whole algorithm is \( O(n \log n) \).

A better data structure makes use of the following observation: We insert the elements in order, first 1, then 2, then 3, and so on. So the largest elements are also the LAST to be INSERTED. Thus, we can use a Stack, since the top of the stack is the last to be inserted, which is also the largest. This gives the following version:

**LastLargerElement(A[1..n]):**

1. \( B[1] \leftarrow 0 \).
2. Initialize a stack \( S \); \( Push.S(1) \).
3. For \( I = 2 \) TO \( n \) do:
   4. \( J \leftarrow Top.S \).
   5. While \( J > 0 \) and \( A[J] \leq A[I] \) do:
      6. \( Pop.S \).
      7. IF \( Top.S \neq NIL \) THEN \( J \leftarrow Top.S \).
      8. ELSE \( J \leftarrow 0 \).
   9. \( B[I] \leftarrow J \).
10. \( Push.S(I) \).
11. Return \( B \).

The only difference is that all operations are now constant time, which means the total time will be \( O(n) \).

**Divide-and-Conquer Recurrence: 10 points** Consider the following recursive algorithm. Its input is an array of positive integers, \( A[1..n] \). The goal is to output the ordered list \( 1 \leq J_1 \leq J_2 \leq \ldots \leq J_k = n \) of unblocked array positions, i.e. those \( 1 \leq J \leq n \) for which \( A[J] > A[I] \) for all \( J < I \leq n \). (Position \( n \) is always considered unblocked, since there are no such \( I \).)

**Unblocked(A[1..n])**

1. IF \( n = 1 \) return a list containing 1.
2. \( L_1 \leftarrow Unblocked(A[1..[n/2]]) \).
3. \( L_2 \leftarrow Unblocked(A[[n/2] + 1..n]) \).
4. While \( L_1 \) is not empty and \( A[\text{Tail}L_1] \leq A[\text{Head}L_2] \) do : delete the tail of \( L_1 \).
5. Return \( L_1 \) appended with \( L_2 \).

Give a recurrence for the time \( T(n) \) taken by the above algorithm. Use the recurrence to give a time analysis up to order. Be sure to justify all of your answers by referring to the algorithm description.

We make two recursive calls to arrays of size \( n/2 \), each returning lists of at most \( n/2 \) elements. Thus, the while loop takes time at most \( O(n) \), since each iteration we delete one of at most \( n \) elements from a list. This gives the recurrence \( T(n) = 2T(n/2) + O(n) \), which fits the main recurrence theorem with \( a = 2, b = 2, k = 1 \). Since \( 2 = 2^1 \), we are in the steady-state case, so \( T(n) \in O(n^b \log n) = O(n \log n) \).