CSE 101 Homework 1

Background (Order and Recurrence Relations), correctness proofs, time analysis, and speeding up algorithms with restructuring, preprocessing and data structures.
Due Thursday, April 23
100 points total = 10 %

Solve each problem. For algorithm problems, if the problem only specifies that you need to give a proof of correctness, then no time analysis is required. If it specifies that you need to give an efficient implementation, then you do not need to give a correctness proof for the basic strategy (just explain why your version actually carries out the strategy). If it says to do both, or doesn’t specify what parts you need, you need to give both a proof of correctness and time analysis.

Order (5 points each)
1. Is \( \log(n^2) \in \Theta(\log n) \)? Why or why not?
Yes, \( \log(n^2) = 2 \log n \), so \( 2 \log n \leq \log(n^2) \leq 2 \log n \) for each \( n > 2 \).

2. If Alg1 takes time \( \Theta(n^2) \) and produces an output of length \( n \log n \), and Alg2 takes time \( \Theta(n^2) \), what is the time complexity of an algorithm that runs Alg1 and then runs Alg2 on the output of Alg1? Explain your answer.

Let \( n' \) be the size of the input to Alg2, which is \( n' = n \log n \) by the problem statement. The time Alg2 takes on this input is \( \Theta((n')^2) = \Theta((n \log n)^2) = \Theta(n^2(\log n)^2) \). Since this is greater than the time Alg1 takes on an input of length \( n \), \( \Theta(n^2) \), the total time is \( \Theta(n^2(\log n)^2) \).

3. Is \( n! \in \Theta(n^n) \)? Why or why not?

No, \( n! = 1 \times 2 \times 3 \times ... \times n \). Upper bounding the first \( n/2 \) terms by \( n/2 \) and the last \( n \) terms by \( n \), \( n! \leq (n/2)^{n/2}(n)^{n/2} = n^n/2^{n/2} \). This is less than \( cn^n \) whenever \( n \geq 2 \log_2 1/c \), so for any \( c > 0 \), \( n! < cn^n \) for sufficiently large \( n \). Thus, \( n! \in o(n^n) \), and is not in \( \Theta(n^n) \).

4. If on a graph with \( n \) nodes and \( m \) edges, Alg1 takes time \( \Theta(nm) \) and Alg2 takes time \( \Theta(n^2 \log n) \), when is it better to use Alg1? Explain.

The break even point is when \( m = \Theta(n \log n) \), but there is one trick to this question. There are constants \( c_1, c_2, c_3, c_4 \) so that Alg1 takes time at least \( c_1 nm \) and at most \( c_2 nm \), and Alg2 at least \( c_3 n^2 \log n \) and at most \( c_4 n^2 \log n \). Then if \( m > c_4/c_3 n \log n \) then it will definitely be better to use Alg1. If \( m < c_3/c_2 n \log n \) then it will definitely be better to use Alg2. Since you don’t know the constants, you can just conclude that if \( m \) grows larger than any constant times \( n \log n \), you should use Alg1, if it is \( o(n \log n) \) you should use Alg2, and otherwise it needs to be determined experimentally.
**Top \( k \) in an array (20 pts)** You are given an array \( A[1..n] \) of \( n \) distinct integers and an integer \( k \) with \( 2 \leq k < n \). You wish to return the \( k \) largest array elements. Design and analyze an algorithm for this problem that runs in time \( O(n \log k) \). Be sure to give a correctness proof and proof of the time analysis.

We’ll use an algorithm based on the following strategy: We keep track of the set \( S \) of the \( k \) largest elements in the sub-array \( A[1..I] \). Initially, \( I = k \) and \( S = \{A[1],..A[k]\} \). To update it to include \( A[I+1] \), we need to see if \( A[I+1] \) is one of the top \( k \) elements in the set \( S \cup \{A[I+1]\} \).

Since there are \( k + 1 \) elements in this set total, \( A[I+1] \) is in the top \( k \) if it is larger than the smallest element of \( S \). So we compare \( A[I+1] \) to the smallest element of \( S \). If it is smaller, we leave \( S \) alone, if not we replace the smallest element of \( S \) with \( A[I+1] \). When \( I = n \), we return \( S \).

We prove by induction that at all times \( I, k \leq I \leq n \), \( S \) really is the largest \( k \) elements of \( A[1..I] \).

Initially, for \( I = k \), \( S \) is all \( k \) elements of \( A[1..k] = A[1..I] \).

Assume coming into the \( I \)’th iteration that \( S \) is the largest \( k \) elements of \( A[1..I] \). This means that all elements of \( S \) are in the set \( A[1..I] \), the size of \( S \) is \( k \), and for any \( s \in S \) and any element \( A[j] \notin S \), with \( 1 \leq j \leq I \), \( A[j] \leq s \). Let \( s_0 \) be the smallest element of \( S \).

If \( s_0 \geq A[I+1] \), \( S \) does not change. So it is still a subset of size \( k \) from \( A[1..I] \subseteq A[1..I + 1] \). Furthermore, for every \( s \in S \) and \( 1 \leq j \leq I \), \( A[j] \notin S \), \( A[j] \leq s \), and also if \( j = I + 1 \), \( A[j] = A[I + 1] \leq s_0 \leq s \). So we have for any \( s \in S \) and any element \( A[j] \notin S \), with \( 1 \leq j \leq I + 1 \), \( A[j] \leq s \). So \( S \) is the \( k \) largest elements of \( A[1..I + 1] \).

If \( s_0 < A[I+1] \), the new set \( S \) becomes \( S = \{s_0\} \cup \{A[I+1]\} \), which still has \( k \) elements from \( A[1..I + 1] \). Furthermore, let \( s \in S \), and \( 1 \leq j \leq I + 1 \), \( A[j] \notin S \). Then \( j < I \), since \( A[I + 1] \notin S \), and hence \( A[j] \leq s_0 \) (since either \( A[I + 1] \) wasn’t in \( S \) before, and so is less than \( s_0 \), or is equal to the only element of \( S \) to be deleted, \( s_0 \).) For any element of \( s \in S \), \( s \geq s_0 \), since \( s_0 \) was the smallest previous element of \( S \) and the only element to be added is \( A[I + 1] \geq s_0 \). So we have \( s \geq s_0 \geq A[j] \) for any such \( s \) and \( j \). So \( S \) is the \( k \) largest elements of \( A[1..I + 1] \).

Thus, in either case, \( S \) remains the \( k \) largest elements in the subsequence. So by induction, this is true at all places \( k \leq I \leq n \), and in particular, is true when the algorithm terminates. Thus, we correctly find the \( k \) largest elements of \( A[1..n] \).

For a fast implementation, note how we use the set \( S \) above: we need to find its smallest element, compare it to \( A[I+1] \) and possibly delete the smallest element and insert a new element. Thus, we can use a min-heap to keep track of \( S \). Note that the size of \( S \) is only \( k \), not \( n \), so we need
have only $k$ places in the heap, not $n$. Thus, operations will be $O(\log k)$ 
time, which is better than $O(\log n)$.

Here is the algorithm:

TopKArray[$A[1..n]$, $k$]

1. Initialize a min heap $S$ of size $k$.
2. FOR $I=1$ TO $K$ do: InsertS($A[I]$)
3. FOR $I = k + 1$ TO $n$ do:
5. FOR $J = 1$ to $K$ do:
6. $B[J] \leftarrow MinS$, DeleteMinS
7. Return $B$.

The last loop puts the elements of $S$ in sorted order in the output array.

As mentioned above, the heap size is $k$, not $n$, so InsertS, DeleteS 
take $O(\log k)$ time. MinS is constant time. We do them each at most $n$ times 
total. This makes the total time $O(n \log k)$.

**Last largest element, 20 points total** Consider the following problem: given 
an array of integers $A[1..n]$, for each $1 \leq I \leq n$, find the last position 
$1 \leq J \leq I$ with $A[J] > A[I]$ (or find 0 if no such $J$ exists), and store it as $B[I]$.

Analyze the obvious algorithm, 5 points

Here is the most obvious algorithm for this problem: (LastLargerElement[$A[1..n]$]: array of integers)

1. FOR $I = 1$ to $n$ do:
   2. $J \leftarrow I - 1$.
4. $B[I] \leftarrow J.$
5. Return $B$

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

The inside loop decrements $J$, which starts at $I - 1 < n$ and ends 
when $J$ reaches 0. Therefore, it iterates at most $n$ times, and each 
iteration is constant time, so is $O(n)$ time overall. It is inside a the 
outer loop, which iterates exactly $n$ times, for a total of $O(n^2)$. If the 
array is sorted from smallest to largest, the inside loop will always
iterate exactly $I$ times, since we’ll never have $A[J] > A[I]$. For the $n/2$ values of $I$, $n/2$ to $n$, this is at least $n/2$ iterations each, for a total of $n^2/4$ iterations. Thus, the algorithm also takes $\Omega(n^2)$ time, so is $\Theta(n^2)$. (Note that it will not usually take this long. If the input array is random, the expected number of decrements until you find a larger value is 2. So the average time on a random array is linear.)

Correctness proof for better strategy Here’s a high-level algorithmic strategy for the same problem:

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

1. $B[1] \leftarrow 0$.
2. Initialize $S$ as $\{1\}$.
3. For $I = 2$ TO $n$ do:
   4. $J \leftarrow$ the largest element of $S$.
   5. While $J > 0$ and $A[J] \leq A[I]$ do:
      6. Delete the largest element of $S$.
      7. IF $S \neq \emptyset$
         THEN $J \leftarrow$ the largest element of $S$
      8. ELSE $J \leftarrow 0$
10. Insert $I$ into $S$.


Prove that this strategy solves the problem. Here’s an outline for how such a proof might go: Let $A[1..n]$ be an input array. Say that $J$ is **unblocked at $I$** if $A[J] > A[K]$ for all $J < K \leq I$. Let $S_I$ be the set of unblocked positions at $I$.

1. Prove that if $J$ is the last larger element to $A[I+1]$, then $J \in S_I$.
2. Prove that if \( 1 \leq J_1 < J_2 < \ldots J_k \leq I \) are the elements of \( S_I \), then \( A[J_1] > A[J_2] > \ldots A[J_k] \).
   If \( A[J_0] \leq A[J_{t+1}] \), then picking \( K = J_{t+1} \) in the definition of unblocked, \( J_t \) would be blocked, and hence not in \( S_I \).

3. Prove the following loop invariant: after the loop \( I, S = S_I \).
   Initially, for \( I = 1 \), both sets are just \( \{1 \} \).
   Assume that \( S = S_I \) after the \( I \)th iteration. We show that after the \( I + 1 \)'st iteration, \( S = S_{I+1} \). In the \( I + 1 \)'st iteration, we first delete members of \( S \) until we reach a member \( J \) with \( A[J] \leq A[I] \). If \( J' \leq I \) were blocked before, it is still blocked by the same value \( K \) at \( I + 1 \). If \( J' \in S_I \) was deleted in this iteration, \( A[J'] \leq A[I + 1] \) and so is blocked by \( K = I + 1 \), and is in neither \( S \) nor in \( S_{I+1} \). If \( J' \) is in \( S_I = S \) at the start of the iteration and not deleted, and \( J \) is the largest value of \( S \) that is not deleted, then \( A[J'] \geq A[J] > A[I + 1] \) by the previous claim. Therefore, since \( A[J'] \geq A[K] \) for \( J' < K \leq I \), and \( A[J'] > A[I + 1] \), \( J' \) is unblocked at \( I + 1 \), and \( J' \) is in both \( S \) and \( S_{I+1} \) at the end of the \( I + 1 \)'st iteration. Finally, for \( J = I + 1 \), we insert \( J \) into \( S \) at the end of the loop, and \( I + 1 \) is trivially unblocked (because no potential blocking positions \( K \) exist). Therefore, \( S = S_{I+1} \) at the end of the iteration.

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).
   In each iteration, we assign \( B[I] \) the largest value \( J \) of \( S \) with \( A[J] > A[I] \), or 0 if no such values exist. Let \( J' \) be the last larger value than \( I \). By claim 1, \( J' \in S_I \). By claim 3, \( S_I = S \) in this iteration. By definition, \( A[J'] > A[I] \) and no larger value of \( J' \) has this property, and \( A[J'] > A[I] \) and no larger value of \( J \in S \) has this property. Therefore, \( J = J' \). If no such larger value exists, no such value exists in \( S \), so all elements of \( S \) will be deleted, and \( B[I] = 0 \) as required.

Efficient versions of algorithm, 5 pts
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 pseudocode 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.

We need to repeatedly find the largest element of \( S \), possibly delete it, and insert \( I + 1 \). A straightforward data structure to use in a max heap of array positions \( J \). The algorithm becomes: LastLargerElement(\( A[1..n] \)): 
1. $B[1] \leftarrow 0$.
2. Initialize a maxheap of size $n$, $S$ to include $\{1\}$.
3. For $i = 2$ TO $n$ do:
4. While $S$ is not empty and $A[\text{Max}S] \leq A[i]$ do: $\text{DeleteMax}S$
5. IF $S \neq \emptyset$
6. THEN $J \leftarrow \text{Max}S$.
7. ELSE $J \leftarrow 0$
9. $\text{Insert}S(\ i)$.
10. Return $B$

Each element $i$ gets inserted into $S$ exactly once, and so deleted at most once. Since every iteration of the while loop in line 4 deletes an element from $S$, it is iterated at most $n$ times, each iteration taking at most $\log n$ time for the DeleteMax heap operation. The rest of the FOR loop is time $O(\log n)$, the Insert being the only non-constant operation. Thus, the total time of the FOR loop is $O(n \log n)$, both from the while and from the rest. The rest of the algorithm is linear time, so the total time is $O(n \log n)$.

A cleverer method is to observe that we insert elements into $S$ in increasing order, and then always delete the largest elements which is the last inserted. Thus, we can use a stack to implement $S$. If we do this, it becomes:

$\text{LastLargerElement}(A[1..n])$:
1. $B[1] \leftarrow 0$.
2. Initialize a stack of size $n$, $S$ to include $\{1\}$.
3. For $i = 2$ TO $n$ do:
4. While $S$ is not empty and $A[\text{Top}S] \leq A[i]$ do: $\text{Pop}S$
5. IF $S \neq \emptyset$
6. THEN $J \leftarrow \text{Top}S$.
7. ELSE $J \leftarrow 0$
9. $\text{Push}S(\ i)$.
10. Return $B$

The only changes to the previous analysis are that heap operations are replaced by constant time stack operations, making the total time $O(n)$.

**Connected components, 20pts** An undirected graph $G$ can be partitioned into connected components, where two nodes are in the same connected component if and only if there is a path connecting them. Design and
analyze an efficient algorithm that computes the connected components of a graph G given in adjacency list format. Be sure to give a correctness argument and detailed time analysis. You can use algorithms from class as a sub-procedure, but be sure to use the claims proven about them carefully. A good algorithm has time approximately $O(n + m)$ where the graph has $n$ nodes and $m$ edges.

We’ll use the following strategy: Pick an arbitrary node, $x$. Since the graph is undirected, the connected component of $x$ is the set of all nodes reachable from $x$. We can use a graph-search (e.g., depth-first search) to find this component. Then we need to find a node not in the current components and repeat. Let $U$ be the set of “unvisited nodes”.

ConnectedComponents($G=(V,E)$):

1. $U \leftarrow V$, $S \leftarrow \emptyset$
2. Initialize $Components$ as an empty list of lists of vertices.
3. While $U \neq \emptyset$, do:
   4. Pick $x \in U$.
   5. $C \leftarrow GraphSearch(G, x)$.
   6. $U \leftarrow U \setminus C$
   7. Append $C$ to $Components$.
8. Return $Components$.

We prove that any version of this strategy correctly finds the connected components of $G$. First, each iteration, we pick some $x \in V$ and return $C$, the set of all the nodes $u$ reachable from $x$ via any path in $G$ (by the correctness of $GraphSearch$, proved in class). Since the graph is undirected, the same path in reverse is a path from $u$ to $x$. So all nodes in $C$ are in the same component as $x$. Conversely, any node $u$ in the same component as $x$ is reachable from $x$, and hence in $C$. So $C$ is a connected component of $G$.

So every set $C$ in $Components$ is a connected component of $G$.

Second, we must show the algorithm finds all connected components. Let $C$ be any connected component, and let $x$ in $C$. Then since $x \in U$ initially, there must be some iteration where $x$ is deleted from $U$. Thus, $x \in C'$ for some $C'$ in $Components$, but as above $C'$ is a connected component of $G$, and nodes are in exactly one connected component. Hence, $C = C'$.

To come up with an efficient version, we can use a doubly-linked list to represent $U$. Then the algorithm becomes:

ConnectedComponents($G=(V,E)$):

1. Initialize a doubly linked list $U$ of all nodes in $V$. 

7
2. Initialize \textit{Components} as an empty list of lists of vertices.

3. While $U \neq \emptyset$, do:
   4. $x \leftarrow U\text{.head}.$
   5. $C \leftarrow \text{GraphSearch}(G, x).$
   6. \text{Foreach} $u \in Cdo$ : Delete$U(u)$
   7. Append $C$ to \textit{Components}.
   8. Return \textit{Components}.

For the time analysis, we need to refer to the lemmas proved in the time analysis for Graph Search. We showed that the main loops in graph search from $x$ take time proportional to the sum of all degrees of vertices $u$ deleted from the frontier set $F$, which is the same as all the vertices reachable from $x$. In other words, when you search from a node $x$, the time taken is proportional to the number of nodes and edges in $x$’s connected component. Since every node and edge is in one connected component, the total time is the number of nodes and edges in $G$ itself (\(O(n + m)\)). Since every node is deleted from $U$ exactly once, and doubly linked list deletion is constant time (changing predecessor and successor pointers of the predecessor and successor of the node), the list operations are \(O(n)\) time total. Thus the entire time is \(O(n + m)\).

**Implementation-20 points** Implement a naive \(O(n^2)\) time sorting algorithm (such as bubble sort) and heap-sort. You can use heaps from a standard library to implement heap-sort. Plot their performance on random arrays of $n$ integers with values between 1 and $n$, for $n = 2^6, 2^8, 2^{10}, 2^{12}, 2^{14}, 2^{16}$. Plot their performance on a log-log scale. Is heap-sort always better than bubble-sort? Why or why not?

See other file.