Background (Order and Recurrence Relations), correctness proofs, time analysis, and speeding up algorithms with restructuring, preprocessing and data structures.
Due Friday, Oct 13
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 (10 points each) 1. Is it always the case that, for every positive integer valued functions $f$ and $g$ with $f(n) \in O(g(n))$, that $g(n) + f(n) \in \Theta(g(n))$? If so, give a proof; if not, give a counter-example.
2. Is it always the case that for every positive integer valued function $f$, that $f(n + 1) \in O(f(n))$? If so, give a proof; if not, give a counter-example.

Correctness Proof: 20 points An array of distinct integers $A[1..n]$ is $k$-almost sorted if for each $1 \leq I \leq n$, the $I$’th smallest element is in positions $A[1], ..., A[I + k]$. Here’s a high level algorithm to sort an input which is $k$-almost sorted:

AlmostSorted[$A[1..n], k$]: $B[1..n]$.
2. FOR $J = k + 1$ TO $n + k$ do:
3. IF $J \leq n$ Add $A[J]$ to S.
4. $B[J - k] \leftarrow$ the smallest element of $S$.
5. Delete the smallest element of $S$
6. Return $B$.

Fill in the blanks in the following proof that the above strategy correctly sorts a $k$-almost sorted list $A$.

We begin by proving a simple invariant: Let $S_j$ be the set $S$ after the iteration when $J = j$, and let $S_k$ be the set $S$ is initialized to. We will show that $\{B[t]|1 \leq t \leq j - k\} \cup S_j = \{A[t]|1 \leq t \leq j\}$ for each $1 \leq j \leq n + k$.

The proof is by induction. For the base case, before the loop begins, i.e., when $j = k$, $S = \{A[t]|1 \leq t \leq k\}$ and there are no $1 \leq t \leq k - k$. Thus, $\cup S = S = \{A[t]|1 \leq t \leq k\}$, and the invariant holds.
Assume the invariant holds after the iteration when \( J = j \), i.e., that IV. We wish to prove that, after the iteration, the invariant still holds, i.e., that V. If \( j + 1 \leq n \), then in this iteration we add one element to \( S \), VI and delete the element that becomes \( B \), VII. Thus, \( S_{j+1} = \{ B[1] \leq t \leq j \} \cup S_j \). Then \( B[t] = \{ B[t] \leq j \} \cup S_j \cup \{ X \} = \{ A[t] \leq t \leq j + 1 \} \). If \( j + 1 > n \), then in this iteration, we only delete \( B[j + 1] \) from \( S \). Thus \( B[t] = \{ B[t] \leq j + 1 \} \cup S_{j+1} \) is the same as \( \{ XI \} \) which by the induction hypothesis is \( \{ A[1], \ldots, A[n] \} \). So in either case, the invariant still holds after the \( j + 1 \)st iteration, so by induction it holds for all iterations.

We now prove by strong induction that each \( B[t] \) is the \( t \)'th smallest element of \( A \). Assume that for all \( 1 \leq t \leq m \), \( B[t] \) is the \( t \)'th smallest element of \( A \). Let \( b \) be the \( t + 1 \)'st smallest element of \( A \). We need to prove that XII. By the invariant proved above, after the loop when \( J = m + 1 + k \), XIII. By the definition of \( k \) almost sorted, the \( m + 1 \)'st largest element of \( A \) is in the set IV, so \( b \) is either in \( S_{m+k+1} \) or in \( B[1..m+1] \). Since by the strong induction hypothesis, the \( m \) smallest elements of \( A \) are \( \{ XV \} \), this means that \( B[m+1] = b \) or \( b \in S_{m+k+1} \). Assume \( b \neq B[m+1] \). Since the algorithm defines \( B[m+1] \) as the smallest element in \( S \), \( B[m+1] \) is in the array \( A \) by the invariant. This XVII the definition of the \( m + 1 \)'st smallest element of \( A \), since \( b \) is larger than each of \( B[1], \ldots, B[XV] \), all of which are in \( A \). From this contradiction, we must have XIX, which is what we wanted to prove.

Thus, by strong induction, \( B[m] \) is the \( m \)'th smallest element of \( A \) for each XX, so \( B \) is the correctly sorted version of \( A \).

**Data structures and preprocessing: 10 points** Give an algorithm that implements the above strategy for sorting a \( k \)-almost sorted list, specifying data structures and giving a time analysis (in terms of both \( n \) and \( k \)).

**Top \( k \) in an array (10 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.

**DAG:** (20 points) A directed graph has edges going from nodes to other nodes, and we write \( x \rightarrow y \) if there is an edge from \( x \) to \( y \). A directed cycle is a list of edges \( x_1 \rightarrow x_2 \rightarrow x_3 \ldots \rightarrow x_k \rightarrow x_1 \). A DAG (Directed Acyclic
Graph is a directed graph with no directed cycles. Design and analyze an efficient algorithm that determines if a directed graph (given in adjacency list format, where there is an array $N(x)$ of lists of nodes with edges to them from each starting node $x$) is a DAG. A good algorithm has time approximately $O(n + m)$ where the graph has $n$ nodes and $m$ edges.

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?