1. **k-friendliness.** (4 points)
Suppose you model the friendships of students in 161 as an undirected graph \( G = (V, E) \) (i.e. there is an edge between students A and B if they are friends). For any integer \( k \), we say that a group of students, \( S \subseteq V \), is “\( k \)-friendly” if every student in \( S \) has at least \( k \) friends in \( S \). Describe an efficient algorithm which, given the graph \( G \) and an integer \( k > 0 \), finds the largest set \( S \subseteq V \) that is \( k \)-friendly; if no such set exists, then the algorithm should say so. Describe the algorithm, prove its correctness, and state/justify the runtime. **[We are expecting a description of the algorithm, a proof of correctness, and a runtime analysis.]**

**SOLUTION:**

**Algorithm:** We proceed by rounds/iterations; in the \( i \)th round, the remaining graph is \( G_i = (V_i, E_i) \). Initially \( G_1 := G \). For \( i = 1, \ldots \), if there exists a vertex with degree less than \( k \) in \( G_i \), remove it and all its incident edges. If on some \( j \)th iteration all remaining vertices have degree at least \( k \) in \( G_j \), return \( V_j \). If \( G_i \) ever becomes empty, report “no solution”.

**Correctness:** Let \( F^* \) be the maximal \( k \)-friendly set. Two observations:

- (For all \( i \), \( V_i \supseteq F^* \)) We claim that the algorithm never deletes any vertices in \( F^* \). To see this, note that in any \( i \)th round, if \( V_i \supseteq F^* \) and some vertex \( v \in V_i \) has degree less than \( k \) in \( G_i \), then it cannot be part of any \( k \)-friendly subset of \( G_i \).

- (If we halt on round \( j \), \( V_j \subseteq F^* \)) When we terminate on some \( j \)th iteration, \( V_j \) contains only the vertices of the solution \( F^* \). This is because we halt precisely when every vertex in \( V_j \) has degree at least \( k \) in \( G_j \); therefore \( V_j \) is a \( k \)-friendly set, so \( V_j \subseteq F^* \).

In conclusion, in the last round \( j \), we have \( V_j = F^* \).

**Running time:** A naive upper bound is \( O(n^2) \), since in \( O(n) \) time we can find a vertex with degree less than \( k \) if one exists; this happens at most \( n \) times. We also delete at most \( m \) edges in total, which takes \( O(m) \) time.

We can improve this to \( O(n + m) \) as follows: maintain a queue \( D \) of vertices to be deleted (initially containing all vertices whose degree in \( G \) is less than \( k \)), and for each \( v \in V \) record its current degree. While \( D \) is nonempty, pop a vertex \( v \) from it, delete \( v \) and its incident edges from the graph, decrement the degrees of \( v \)'s neighbors, and enqueue any of them whose degree falls below \( k \).

2. **Transportation networks.** (5 points)

Given a set of \( n \) cities, we would like to build a transportation system such that there is some path from any city to any other city. There are two ways to travel: by driving or
by flying. Initially all of the cities are disconnected. It costs \( c_{i,j} \) to build a road between city \( i \) and city \( j \). It costs \( a_i \) to build an airport in city \( i \). For any two cities \( i \) and \( j \), we can fly directly from \( i \) to \( j \) if there is an airport in both cities. Give an efficient algorithm for determining which roads and airports to build to minimize the cost of connecting the cities. Here, “connecting the cities” means that there should be some way to get from any city to any other.

You algorithm should take as input the costs \( c_{i,j} \) and \( a_i \), and return a list of roads and airports to build. It should run in time \( O(m \log(n)) \).

[We are expecting a description or pseudocode of your algorithm and a short informal explanation of why it is correct. You may (and, hint, you may wish to) call any algorithm we have seen in class.]

**SOLUTION:**

Consider a graph \( G \) constructed as follows. We have a vertex \( v_i \in V \) for \( i = 1, \ldots, n \), one for each of the \( n \) cities, and there is an edge between \( v_i \) and \( v_j \) with cost \( c_{i,j} \). Consider also the graph \( G_{sky} \), which is the same as \( G \) and where we additionally add a vertex \( \text{sky} \) representing the sky. We add an edge between \( v_i \) and \( \text{sky} \) with cost \( a_i \). Now our algorithm is:

```python
planTransit(costs c_{i,j}, a_i):
    generate the graph G as above
    T = KRUSKAL(G) // or any MST algorithm
    T_sky = KRUSKAL( G_sky )
    if cost(T) < cost(T_sky):
        T_actual = T
    else:
        T_actual = T_sky
    airports = []
    roads = []
    for i = 1,...,n:
        if {v_i, air} is an edge in T_actual:
            airports.add(i)
    for j =1,...,n:
        if {v_i, v_j} is an edge in T_actual:
            roads.add( {i,j} )
    return airports, roads
```

Next, we argue that this is correct. There are two cases: either the optimal solution uses an airport, or it does not. If the optimal solution does not include an airport, then the optimal solution is given by an MST on \( G \); that is, the minimum cost solution using only roads. In contrast, if the optimal solution does use an airport, then the cost is given by an MST on \( G_{air} \). The pseudocode above returns the better of the two.

3. **Placing receivers.** (9 points)
Suppose there are \( n \) transmitters fixed in place along a linear track. The \( i \)’th transmitter has communication range \([a_i, b_i]\), for \( a_i \leq b_i \). That is, any receiver placed within the range \([a_i, b_i]\) can receive signals from the \( i \)’th transmitter. Assume that the transmitters are sorted by the right endpoint of their communication range: that is, if \( i < j \), then \( b_i \leq b_j \).

We want to pick a set of points on the track to place receivers such that we can receive signals from every transmitter while minimizing the number of receivers necessary. That is, we want to find a minimum set of points \( S \) on the line such that for every transmitter \( i \), there is some receiver \( s \in S \) such that \( a_i \leq s \leq b_i \).

In this problem, we will design an algorithm that finds the minimum set \( S \) of receiver locations in expected time \( O(n) \), and prove that it is correct.

Consider the greedy algorithm which works as follows: we place receivers one at a time. At each step, suppose that \( i^* \) is the smallest \( i \) so that transmitter \( i \) cannot be heard by any receiver placed so far, and place a receiver at \( b_{i^*} \). Continue placing receivers in this way until all the transmitters can be heard.

(a) (2 points) Based on this English description, write pseudocode to implement the algorithm in time \( O(n) \). Assume the input to your algorithm is two arrays, \( a \) and \( b \), which contain the values of \( a_i \) and \( b_i \) in the order described above.

[We are expecting detailed pseudocode, and an informal justification of the running time.]

(b) (7 points) Prove that this algorithm is correct (in terms of both legality and optimality), following the outline below. We recommend induction on \( t \), using a greedy exchange argument.

i. (2 points) State your inductive hypothesis.
ii. (1 point) Prove the base case.
iii. (3 points) Prove the inductive step.
iv. (1 point) Finish the argument. That is, once the induction argument is complete, show that this implies that the algorithm is correct.

[We are expecting: for i, a statement of an inductive hypothesis. For ii,iii,iv, we are expecting a formal proof, including a statement of what it is you are proving.]

SOLUTION:

(a) The pseudocode is:

\[
\text{placeReceivers}(a,b): \quad \text{\( a \) and \( b \) are arrays of } a_i \text{ and } b_i\n\]
\[
\text{biggestReceiver} = -\text{Infinity} \\
\text{receivers} = [] \\
\text{for } i = 1, \ldots, n: \\
\hspace{1em} \text{if } a[i] \leq \text{biggestReceiver} \text{ and } \text{biggestReceiver} \leq b[i]: \\
\hspace{2em} \text{continue}
\]
else:
    receivers.append(b[i])
    biggestReceiver = b[i]

return receivers

This runs in time $O(n)$ because the outer loop is over $n$ things, and inside this loop we do $O(1)$ work to update the list receivers and the variable biggestReceiver.

(b) i. The inductive hypothesis is: after the $i$'th iteration of the for loop, the set receivers is contained in an optimal set $S^*$.

ii. The base case is when $i = 0$. In this case, the inductive hypothesis says that after 0 iterations of the for loop, the set receivers is contained in an optimal set $S^*$. Since receivers is empty at this point, the base case holds trivially.

iii. To prove the inductive step, we must show that if the base case holds for $0 \leq i \leq n-1$, then it holds for $i+1$.

Optimality Claim: Suppose that after the $i$'th iteration of the for loop, the set receivers is contained in an optimal set $S^*$. Now in the $i+1$'st iteration, two things could happen. The first is that transmitter $i+1$ is already covered, in which case receivers does not change, and so the inductive hypothesis holds at step $i+1$.

On the other hand, suppose that transmitter $i+1$ is not covered already. Since receivers does not cover transmitter $i+1$, there must be some $p \in S^* \setminus$ receivers that covers transmitter $i+1$. Let $p$ be the smallest such element. Notice that this implies that $p \leq b_{i+1}$, since $p \in [a_{i+1}, b_{i+1}]$ covers $b_{i+1}$.

Consider the set $S' = S^* \cup \{b_{i+1}\} \setminus \{p\}$. Notice that the size of $S'$ is the same as the size of $S^*$.

Legality Claim: $S'$ covers all of thetransmitters. The only intervals we need to worry about are those that are covered by $p$ and by no other point in $S^*$. Suppose that $[a_k, b_k]$ is covered by $p$ and by no other point in $S^*$. We need to show that $a_k \leq b_{i+1} \leq b_k$, because then this interval will be covered by the new transmitter at $b_{i+1}$.

- First, since $p \leq b_{i+1}$ and $p$ covers $[a_k, b_k]$, we have $a_k \leq p \leq b_{i+1}$.
- Second, for all $j < i + 1$, $[a_j, b_j]$ is covered by receivers by construction. In particular, $[a_j, b_j]$ is covered by something other than $p$, so we must have $k \geq i + 1$. But then we have $b_k \geq b_{i+1}$, since they are sorted.

Together, these two points establish that $a_k \leq b_{i+1} \leq b_k$, which is what we needed to show.

Thus, $S'$ is also an optimal set of covering transmitter locations, and $b_{i+1} \in S'$. This establishes the inductive hypothesis for $i+1$.

iv. Finally, at for $i = n$, the inductive hypothesis reads: after the $n$'th iteration of the for loop, the set receivers is contained in an optimal set $S^*$. Since by construction receivers covers all of the transmitters, and it is contained in $S^*$, we have $|receivers| \leq |S^*|$, and hence receivers also has optimal size.

4. Warmup with MSTs. (7 points)
In all of the following questions, assume that $G$ is an undirected, connected graph on $n$ nodes with distinct, positive edge weights. Also assume that $G$ is simple (no self-loops and no multi-edges) in all parts.

(a) (1 point) Consider the edges of $G$ sorted in increasing order of weight and place the first $n - 1$ edges into a set $S$. That is, $S$ is the set of $n - 1$ edges with the smallest weights.

Prove or disprove: $S$ is a minimum spanning tree in $G$ (assume $n \geq 2$)

[We are expecting either a formal proof or a counterexample.]

(b) (2 points) Prove or disprove: There is a unique minimum spanning tree in $G$.

[We are expecting either a formal proof or a counterexample.]

(c) (4 points) Let $T$ be a minimum spanning tree of $G$. Let $e$ be some edge in $G$ (which may or may not belong to $T$). Let us obtain a new graph $G'$ from $G$ by preserving everything the same except that we decrease the weight of $e$ (but assume that the weights of all edges in $G'$ are still distinct). Design an algorithm that can find a minimum spanning tree $T'$ of $G'$ in time $O(n)$, given $G$, $T$, $e$ and its newly decreased weight $w$. Prove the correctness and analyze the runtime.

[We are expecting an algorithm (description of one is fine), a proof of correctness, and a runtime analysis.]

SOLUTION:

(a) Consider a graph with four vertices \{a, b, c, d\} and edges $w(a, b) = 1, w(a, c) = 2, w(b, c) = 3, w(a, d) = 4$. $S$ consists of the first three edges which forms a cycle, and thus is not a spanning tree.

(b) There is a unique min-cost spanning tree.

We use the cycle property as lemma: the most expensive edge on any cycle does not belong to MST (when edge weights are distinct).

Let $C$ be any cycle in $G$ and $e$ be the most expensive edge on $G$. Suppose $T$ is any spanning tree containing $e$ (we show that $T$ is not a MST then). If we delete $e$ from $T'$, then $T'\setminus\{e\}$ partitions the vertices into two connected components (two trees), $(S, V \setminus S)$; if $e = (u, v)$, let $S$ be the set of nodes connected to $u$ in $T'\setminus\{e\}$ and $V \setminus S$ be the set of nodes connected to $v$. Since $C\setminus\{e\}$ is a path from $u$ to $v$ (by definition of a cycle), there must be some edge $e'$ crossing the cut; in particular, $w(e) > w(e')$ by assumption, and let $T' = T\setminus\{e\} \cup \{e'\}$ (then $T'$ has smaller cost than $T$). We claim that $T'$ is a spanning tree. We already know that $S$ and $V \setminus S$ are trees (from $T$), and for every node $x \in S$ and $y \in V \setminus S$, there is a path from $x$ to $u$ (in $S$) and $y$ to $v$ (in $V \setminus S$), and we use $e'$ to create a path between $x, y$. This shows $T'$ is a spanning tree.

To prove the main part: let $T$ be any MST in $G$. We show that $T$ is unique. For any edge $e' \notin T$, $e' \cup T$ contains a cycle $C$ with $e'$ in it; $e'$ must be the most expensive edge on this cycle - otherwise, if there is another edge $e''$ on $C$ that is more expensive, then $T$ cannot be MST by the lemma, which is a contradiction. This applies for every edge $e'$ not in $T$, and we conclude that every edge $e' \notin T$ does not belong to any MST. Thus $T$ is the only MST in $G$. 

5
(c) We use the cycle property as lemma: the most expensive edge on any cycle does not belong to MST.

Let \( T \) be an MST of \( G \) as stated, and let \( e \) be some edge we are reducing the weight of (recall \( T \) is unique from previous part). Consider any edge \( e' \not\in T \); since \( T \cup \{e'\} \) induces a cycle containing \( e' \), \( e' \) must be the most expensive edge on this cycle (or it contradicts the lemma). Note that this holds for all \( e' \not\in T \).

Now we reduce the weight of \( e \); for every \( e' \not\in T \) with \( e' \neq e \), \( e' \) remains to be the most expensive edge on the cycle in \( T \cup \{e'\} \) because we are reducing the weight of \( e \) which does not affect \( e' \) being the most expensive. Therefore, in the new graph, any MST (there is a unique one due to part (a)) cannot contain any \( e' \not\in T \) with \( e' \neq e \). In particular, if \( e \in T \), then \( T \) remains to be an MST in the new graph, and if \( e \not\in T \), then an MST must contain edges from \( T \cup \{e\} \). In the latter case, \( T \cup \{e\} \) creates a cycle containing \( e \); if \( e = (u, v) \), the cycle contains \( e \) and the \( u-v \) path on \( T \). By the cycle property, the most expensive edge in this cycle cannot be in an MST of the new graph.

With this proof, the following algorithm’s correctness follows:

Given \( T, G \) and \( e \) and its new weight \( w \), check if \( e \in T \) or not (we can just scan the edges of \( T \) in linear time). If \( e \in T \), just output \( T \). If \( e \notin T \), let \( e = (u, v) \). Using DFS\((u)\) on \( T \), find the \( u-v \) path in \( O(n) \) time (since \( T \) contains \( n-1 \) edges, this runs in \( O(n) \) time). Scan the \( u-v \) path on \( T \) to find the most expensive edge \( e'' \) on this path in \( O(n) \) time; if \( w(e'') > w \) then we output \( T' = T \cup \{e\} \setminus \{e''\} \) a answer, and if \( w(e'') < w \) then we output \( T \).

5. **Min Gradespan (4 points)** An analog of the following problem arises when you are trying to assign jobs to servers; while the greedy algorithm will not be optimal, it comes pretty close.

Suppose we have \( n \) exams to grade, and there are \( k \) Course Assistants (CAs) who will be doing the grading. Suppose can tell the exact amount of time it will take to grade a given exam (from glancing at the handwriting and text density), and all the CAs would take this exact amount of time to grade that exam. I want to divide the set of \( n \) exams among the CAs so as to minimize the maximum amount of time it will take any of the CAs to grade (i.e. we will all wait until the last CA is done grading their pile, and I want to minimize the total time from when we all start grading, until this last CA is done.)

Suppose I start with the stack of exams, and want to divide the exams into the \( k \) stacks by stepping through the pile of exams just once—namely, when I look at the \( i \)th exam, I will immediately know how long it takes to grade, and then assign it to one of the \( k \) CA piles. Suppose I do this by keeping track of the total grading time of each stack, and assigning the next exam to the stack that currently has the shortest grading time.

(a) (1 point) Describe a concrete instance of this problem involving \( n = 3 \) exams and \( k = 2 \) CAs that illustrates that this algorithm will not always result in the optimal allocation. Specifically, give a list of the three grading times such that the greedy algorithm will result in one of the two CAs spending longer than necessary, and
describe both the optimal allocation of the exams to the two piles, and the allocation that the greedy algorithm will choose.

[We are expecting an example. You can be as specific with the example as you want.]

(b) (3 points) Prove that the allocation given by the greedy algorithm will be suboptimal by at most a factor of 2. Namely, if there exists an allocation of the n exams into k piles such that all CAs can finish grading within time \( t_{\text{opt}} \), prove that in the allocation chosen by the greedy algorithm, the maximum time will be at most \( 2 \cdot t_{\text{opt}} \). [Hint: what can you say about the amount of grading time that the longest-working CA would have had, if the last exam did not exist?]

[We are expecting a formal proof.]

(c) (0 points, Food for thought) How much can you improve this factor-of-two ratio of greedy and opt (i.e. this factor of 2 “competitive ratio” of the greedy algorithm) if you sort the exams in decreasing order of grading time before running the greedy algorithm?

[We are not expecting anything.]

SOLUTION:

(a) Imagine there are 3 exams with grading times 10 minutes, 10 minutes, and 20 minutes, in that order, and 2 CAs. The optimal grading time is 20 minutes, but the greedy algorithm will assign each of the CAs one of the 10 minute exams, and then give one of them the 20 minute exam, which means the overall time is 30 minutes, for a ratio of 3/2.

(b) Let \( k \) denote the number of CAs. Consider the longest-working CA based on the greedy assignment, and let \( a \) denote the amount of time required by the last exam that they were assigned.

Let \( b \) denote the sum of the times of all other exams. \( OPT \geq \max(a, \frac{a+b}{k}) \), since someone needs to grade the exam that takes time \( a \), and the total work is \( a+b \), so someone must do at least a \( 1/k \) fraction of that work. The amount of work assigned to the longest-working CA by the greedy algorithm is at most \( a + \frac{b}{k} \), since prior to being assigned the exam with time \( a \), they had the shortest stack, and hence the height of that stack could have been at MOST a \( 1/k \) fraction of the work assigned up to that point, which is bounded by \( b \).

To conclude, note that if \( a \geq \frac{b}{k} \), then clearly

\[
\text{Greedy} \leq a + \frac{b}{k} \leq 2a \leq 2.OPT.
\]

If \( a < \frac{b}{k} \) then

\[
\text{Greedy} \leq a + \frac{b}{k} \leq 2 \cdot \frac{b}{k} \leq 2 \cdot \frac{a+b}{k} = 2.OPT.
\]

6. How is the course so far? (0.5 points extra credit)

Please complete the poll at https://goo.gl/forms/8cKzJ2u08evIc6EM2.