Question 1

Part a)

Required: Prove a tight bound on the worst-case runtime complexity of an individual `Insert` operation on a partially sorted array that contains n integers.

Big-Oh Bound: Consider a partially sorted array that contains n integers. After appending a new item to the array, we increment `size` by 1 so that `size` $= n + 1$. In the worst-case, this causes `size` $\geq 2 \cdot$ `sortedsize` leading us to execute `insertionsort`. Since $n + 1 =$ `size` $\geq 2 \cdot$ `sortedsize`, we have that `sortedsize` $\leq \lfloor \frac{n+1}{2} \rfloor$. And we know `sortedsize` ≥ 0. Hence, we have that $0 \leq i \leq \lfloor \frac{n+1}{2} \rfloor$ for line 2. Since we are considering an upper bound, we will consider $i = 0$, so that we have at most `size` $- i = n + 1$ many iterations of the while loop on line 3. Note, we may have less iterations given that our array is partially sorted, but we are looking for an upper bound. For each such i from 0 to n, we let $j = i$, and have at most $j = i$ many iterations of the while loop on line 5, and hence at most $j = i$ executions of line 6.

In total we have at most $\sum_{i=0}^{n} i = \frac{n(n+1)}{2}$ executions of line 6. And recall we had appended 1 element to our array at the start. Hence, we have $\mathcal{O}(n^2)$ as an upper bound.

Omega Bound: Consider a family of inputs, one for each $n \in \mathbb{N}$ as follows. For each $n \in \mathbb{N}$, let `arr` contain n integer elements such that `arr` $= \left[\lfloor \frac{n}{2} \rfloor - 1, \lfloor \frac{n}{2} \rfloor, ..., n-1, n, \lfloor \frac{n}{2} \rfloor - 2, ..., 3, 2, 1\right]$ so that the first $\lfloor \frac{n}{2} \rfloor$ elements of `arr` are sorted in nondecreasing order, and the remaining elements are sorted in nonincreasing order.

Now assume we append the integer 0 to `arr`. We will increment `size` so that `size` $= n + 1$. Note, `sortedsize` $= \lfloor \frac{n}{2} \rfloor$, so we get that `size` $\geq 2 \cdot$ `sortedsize`. Hence, we execute `insertionsort`.

Since `arr[sortedsize:]` is nonincreasing with distinct elements, we have that `arr[sortedsize:]` is actually decreasing. Hence, for each i from $i =$ `sortedsize` $= \lfloor \frac{n}{2} \rfloor$ to $i =$ `size` $- 1 = n$, we will have at least $j = i$ many executions of the while loop on line 5, and hence at least $j = i$ many executions of line 6, since `arr[sortedsize:]` is strictly decreasing and each element must be moved to the leftmost part of the array during its iteration of the while loop of line 5.

We have the following number of executions of line 6 when we sum up the above discussion.

$$\begin{aligned}
\sum_{i=\lfloor \frac{n}{2} \rfloor}^{n} i &= \sum_{i=0}^{n} i - \sum_{i=0}^{\lfloor \frac{n}{2} \rfloor - 1} i \\
&= \frac{n(n+1)}{2} - \frac{\lfloor \frac{n}{2} \rfloor (\lfloor \frac{n}{2} \rfloor - 1)}{2} \\
&= \Omega(n^2)
\end{aligned}$$

Hence, we have $\Omega(n^2)$ executions of line 6. We also have 1 step for appending an element 0 to `arr` at the start. Hence, we have a lower bound of $\Omega(n^2)$. Since we have $\mathcal{O}(n^2)$ and $\Omega(n^2)$, we conclude a tight bound of $\Theta(n^2)$.

Part b)

Required: Consider a sequence of k insert operations performed on an initially empty partially sorted array. Use aggregate analysis to prove a tight bound on the amortized complexity per operation in this sequence.

First we will show an upper bound on the worst case sequence complexity ($WCSC$) of k insert operations.

Let $t(i)$ be the number of steps taken for the i-th insert operation where $i \in \{1, ..., k\}$.

For inserts that do not call `insertionsort`, we require only 1 step to append an item to our array.

For inserts that do call `insertionsort`, we require 1 step to append an item to our array and then $\mathcal{O}(i^2)$ many steps for `insertionsort` after our i-th insert. This follows from our runtime analysis in part a).

Since we only call `insertionsort` when `size` $\geq 2 \cdot$ `sortedsize`, we have that we call `insertionsort` on inserts when i is a power of 2. i.e. when $i = 2^j$ for some $j \geq 1$.

Hence, we get the following for $t(i)$.

$$t(i) \leq \begin{cases} 1 + \mathcal{O}(i^2) & \text{if } i = 2^j \text{ for some } j \geq 1 \\ 1 & \text{otherwise} \end{cases}$$

Hence, we have

$$t(i) - 1 \leq \begin{cases} \mathcal{O}(i^2) & \text{if } i = 2^j \text{ for some } j \geq 1 \\ 0 & \text{otherwise} \end{cases}$$

Let $WCSC$ be the worst case sequence complexity for k inserts. Please see the next page.

$$
\begin{aligned}
WCSC &\leq \sum_{i=1}^{k} t(i) \\
&= k + \sum_{i=1}^{k} (t(i) - 1) \\
&= k + \sum_{j=1}^{\lfloor \log(k) \rfloor} t(2^j) \\
&= k + \sum_{j=1}^{\lfloor \log(k) \rfloor} \mathcal{O}((2^j)^2) \\
&= k + \sum_{j=1}^{\lfloor \log(k) \rfloor} \mathcal{O}((2^2)^j) \\
&= k + \sum_{j=1}^{\lfloor \log(k) \rfloor} \mathcal{O}(4^j) \\
&= k + \frac{4}{3}\mathcal{O}(4^{\lfloor \log(k) \rfloor} - 1) \\
&= k + \frac{4}{3}\mathcal{O}(2^{2\lfloor \log(k) \rfloor} - 1) \\
&\leq k + \frac{4}{3}\mathcal{O}(2^{2\log(k)} - 1) \\
&= k + \frac{4}{3}\mathcal{O}(2^{\log(k^2)} - 1) \\
&= k + \frac{4}{3}\mathcal{O}(k^2 - 1) \\
&= \mathcal{O}(k^2)
\end{aligned}
$$

So we have that $WCSC = \mathcal{O}(k^2)$. Now we will find a corresponding lower bound.

Consider the following family of sequences of k insert operations. Our sequence of k inserts is $insert(k), insert(k-1), ..., insert(2), insert(1)$ in that order. This sequence will cause the maximum amount of swaps (i.e. executions of line 6) during each call to `insertionsort` on any i-th insert where $i = 2^j$ for some $j \geq 1$ since we are inserting in strictly decreasing order. And we know that we will have at least $\Omega(i^2)$ steps during these calls.

Let $s(i)$ be the number of steps taken on the i-th insert of our sequence described above where $i \in \{1, ..., k\}$. Hence, we have the following.

$$
s(i) \geq \begin{cases} 1 + \Omega(i^2) & \text{if } i = 2^j \text{ for some } j \geq 1 \\ 1 & \text{otherwise} \end{cases}
$$

Hence, we have

$$s(i) - 1 \geq \begin{cases} \Omega(i^2) & \text{if } i = 2^j \text{ for some } j \geq 1 \\ 0 & \text{otherwise} \end{cases}$$

Hence,

$$\begin{aligned} WCSC &\geq \sum_{i=1}^{k} t(i) \\ &= k + \sum_{i=1}^{k} (s(i) - 1) \\ &= k + \sum_{j=1}^{\lfloor \log(k) \rfloor} s(2^j) \\ &= k + \sum_{j=1}^{\lfloor \log(k) \rfloor} \Omega((2^j)^2) \\ &= k + \sum_{j=1}^{\lfloor \log(k) \rfloor} \Omega((2^2)^j) \\ &= k + \sum_{j=1}^{\lfloor \log(k) \rfloor} \Omega(4^j) \\ &= k + \frac{4}{3}\Omega(4^{\lfloor \log(k) \rfloor} - 1) \\ &\geq k + \frac{4}{3}\Omega(4^{\log(k)-1} - 1) \\ &= k + \frac{4}{3}\Omega(2^{2\log(k)-2} - 1) \\ &= k + \frac{4}{3}\Omega(2^{\log(k^2)-2} - 1) \\ &= k + \frac{4}{3}\Omega(\frac{1}{4}k^2 - 1) \\ &= \Omega(k^2) \end{aligned}$$

Since $WCSC = \mathcal{O}(k^2)$ and $WCSC = \Omega(k^2)$, we get that $WCSC = \Theta(k^2)$.

Hence, we get the following amortized complexity.

$$\begin{aligned} amortized &= \frac{WCSC}{k} \\ &= \frac{\Theta(k^2)}{k} \\ &= \Theta(k) \end{aligned}$$

Therefore, the amortized complexity per operation in a sequence of k insert operations is $\Theta(k)$, as required.

Question 2

Part a)

Consider the following counterexample. Let $\{s_1, s_2, s_3, s_4\}$ be our set of students.

For each $i, j \in \{1, 2, 3, 4\}$ such that $i \neq j$, let

$$f(s_i, s_j) = \begin{cases} 0 & \text{if } i = 4 \text{ or } j = 4 \\ 1 & \text{otherwise} \end{cases}$$

Let T_0, T_1 be any valid partitioning of our set of students into 2 teams. Hence, we know $T_0 \cup T_1 = \{s_1, s_2, s_3, s_4\}$ and $T_1 \cap T_2 = \emptyset$ and $T_0, T_1 \neq \emptyset$.

Since $T_0, T_1 \neq \emptyset$, we must be in one of the following two cases.

Case 1: $|T_0| = |T_1| = 2$.

Without loss of generality, assume $T_0 = \{s_x, s_y\}$ and $T_1 = \{s_z, s_4\}$ where $x, y, z \in \{1, 2, 3\}$.

Let $j = 1$, and let $s_i = s_z \in T_1 = T_j$. Let $s_{i'} \in T_j - \{s_i\}$ be arbitrary. Note, we must have that $s_{i'} = s_4$.

Let $s_k = s_1 \in T_0 = T_{j-1}$. We have that $f(s_i, s_{i'}) = f(s_z, s_4) = 0 < 1 = f(s_z, s_1) = f(s_i, s_k)$.

So $f(s_i, s_{i'}) < f(s_i, s_k)$.

So the negation of property $(*)$ holds.

Therefore, property $(*)$ does not hold.

Case 2: One of T_0, T_1 has cardinality 1, and the other has cardinality 3.

Without loss of generality, assume $|T_0| = 1$ and $|T_1| = 3$.

Assume for the sake of contradiction that $(*)$ holds.

So let $j = 0$ and let $s_i \in T_0 = T_j$. Hence, there exists an $s_{i'} \in T_j - \{s_i\} = \emptyset$. This is a contradiction as the empty set contains no elements.

Hence, property $(*)$ cannot hold.

In either case, we have that $(*)$ does not hold for our example of students, as required.

Please see the next page.

Part b)

We will use the graph ADT. We will also use the union by rank with path compression implementation of the Disjoint Set ADT, along with Kruskal's algorithm.

Consider the following weighted undirected graph $G = (V, E)$.

Let $V = \{s_1, ..., s_n\}$ and $E = \{\{s_i, s_j\} : s_i, s_j \in V \wedge s_i \neq s_j\}$. For each $\{s_i, s_j\} \in E$, let $w(\{s_i, s_j\}) = f(s_i, s_j)$ where w is our weight function and f is our friendship scoring.

We will sort our edges by non-increasing weights in order to apply Kruskal's algorithm. Note that this application of Kruskal's algorithm will consider the most costly edges first. i.e. Kruskal's algorithm will look for a maximum spanning tree.

Consider the following algorithm below.

TeamFinder($\{s_1, ..., s_n\}$)

1: $V \leftarrow \{s_1, ..., s_n\}$
2: $E \leftarrow \{\{s_i, s_j\} : s_i, s_j \in V \wedge s_i \neq s_j\}$
3: **for** each $\{s_i, s_j\} \in E$ **do**
4: $\quad w(\{s_i, s_j\}) \leftarrow f(s_i, s_j)$
5: **end for**
6:
7: $G \leftarrow (V, E)$ ▷ G is a connected, undirected, weighted graph
8: Sort edges of E by weight in non-increasing order.
9: Run Kruskal's algorithm to find a maximum spanning tree T. Also keep track of the last edge $e_0 \in E$ added to T.
10:
11: $A \leftarrow$ the set of vertices belonging to edges in T. ▷ Note, $A = V = \{s_1, ..., s_n\}$
12:
13: $T' \leftarrow T \setminus \{e_0\}$
14: $B \leftarrow$ the set of vertices belonging to edges in T'
15:
16: $T_0 \leftarrow B$ ▷ Note, $|T_2| = n - 1$
17: $T_1 \leftarrow A \setminus B$ ▷ Note, $|T_1| = 1$
18:
19: **return** T_0 and T_1

Please see the next page.

Part c)

We will now explain the correctness of our algorithm.

Lines 1-7 simply constructs a connected, undirected, weighted graph $G = (V, E)$, where V is our set of students, and E is the set of all unordered pairs of students. And the weights of each edge are the friendship scores of that pair of students.

We sort E by non-increasing weights (i.e. non-increasing friendship scores), and then run Kruskal's algorithm to find a maximum spanning tree T.

We also keep track of the last edge e_0 added to T. We let A be the set of all vertices of edges belonging to T. Since T is a maximum spanning tree, we know that $A = V$.

We then let B be the set of all vertices belonging to edges in $T' = T \setminus \{e_0\}$.

Since T was a maximum spanning tree, we know that B contains every vertex of $A = V$, except one vertex which was part of e_0.

We then let $T_0 = B$, and $T_1 = A \setminus B$. Hence, we have $|T_0| = n - 1$ and $|T_1| = 1$.

Want to Show: T_0 and T_1 satisfy our modified property $(*)$.

W know that $|T_1| = 1$. Hence, ii) of $(*)$ trivially holds for $T_j = T_1$.

Now consider $T_j = T_0$. Let $s_i \in T_j = T_0$. We will show that i) holds.

Let $s_{i'}$ be the vertex such that $\{s_i, s_{i'}\}$ is the first edge added to T in our algorithm that includes s_i.

Let $s_k \in T_{1-j} = T_1$. Since $|T_1| = 1$, we have that $T_0 = \{s_k\}$. Consider our tree T prior to adding the edge $\{s_i, s_{i'}\}$.

Since $s_k \in T_1$ was the last vertex to be reached from our maximum spanning tree, we know that s_k was not part of any edge in T prior to adding the edge $\{s_i, s_{i'}\}$.

Hence, s_k and s_i were not part of any edge of T prior to adding the edge $\{s_i, s_{i'}\}$. Hence, Kruskal's algorithm had the option of choosing to add the edge $\{s_i, s_k\}$ instead of $\{s_i, s_{i'}\}$ without creating any cycles. The only reason Kruskal chose $\{s_i, s_{i'}\}$ instead of $\{s_i, s_k\}$ must be that $w(\{s_i, s_{i'}\}) \geq w(\{s_i, s_k\})$.

Since our weights equal their corresponding friendship scores, we have $f(s_i, s_{i'}) \geq f(s_i, s_k)$. Hence, i) holds. Therefore, the modified $(*)$ property holds for T_0 and T_1 given by our algorithm. Therefore, our algorithm is correct, as required.

Part d)

Required: Explain why the worst-case runtime of our algorithm is $\mathcal{O}(n^2 \log n)$.

Note, since G considers all (unordered) pairs of vertices as edges, we have that $|E| = \binom{|V|}{2} = \frac{|V|(|V|-1)}{2}$.

We know that constructing G takes $\mathcal{O}(|V| + |E|) = \mathcal{O}(|V|^2)$ time using an adjacency list and given the fact that $|E| = \binom{|V|}{2} = \frac{|V|(|V|-1)}{2}$.

And sorting the edges of G by weight in non-increasing order (using say mergesort) takes $\mathcal{O}(|E| \log |E|)$ time. And Kruskal's algorithm also takes $\mathcal{O}(|E| \log |E|)$ time.

Note, $\mathcal{O}(|E| \log |E|) = \mathcal{O}(|V|^2 \log(|V|^2)) = \mathcal{O}(2|V|^2 \log |V|) = \mathcal{O}(|V|^2 \log |V|)$ time.

Determining the sets A, B, T', T_0, T_1 all take at most $\mathcal{O}(|E|) = \mathcal{O}(|V|^2)$ time.

Hence, the overall runtime is $\mathcal{O}(|V|^2 \log |V|)$.

Since $|V| = |\{s_1, ..., s_n\}| = n$, we have that the overall runtime is $\mathcal{O}(n^2 \log n)$, as required.

Question 3

Part a)

We will use the graph ADT. In particular, we will use a directed graph. Depending on the instance of our problem, our directed graph may or may not be cyclic. In fact, testing for cycles will be part of our algorithm in part b).

Let $\mathcal{B}$ be a set of observations.

Let $V = \{R_1, ..., R_n\}$ be our set of vertices. Let E be our set of edges defined below.

For each $R_i \to R_j \in \mathcal{B}$ where $i < j$, we have an edge $(R_i, R_j) \in E$ and edges $(R_i, R_k) \in E$ for each k such that $j < k \leq n$. Note, $R_i \to R_j \in \mathcal{B}$ implies that R_i finishes before R_j begins, and hence we have that R_i finishes before R_k begins for each k such that $j < k \leq n$ since each runner starts strictly before the next runner in our sequence of runners.

For each $R_i \sim R_j \in \mathcal{B}$ where $i < j$, we have an edge $(R_j, R_i) \in E$.

Let $G = (V, E)$ be our directed graph which we will use to implement our algorithms in parts b) and c).

Part b)

Consider the following algorithm.

CheckConsistency($\mathcal{B}$)

```
1: Construct the graph G = (V, E) as described in part a).
2: Run DFS (depth first search) on G and construct the DFS tree T.
3: for each edge e ∈ E do
4:     if e is a back edge with respect to the DFS tree T then
5:         return "INCONSISTENT"                    ▷ We have a cycle
6:     end if
7: end for
8: return "CONSISTENT"                              ▷ We have no cycles
```

Explanation of Correctness: We know that a given $\mathcal{B}$ is inconsistent if there exists some $R_i \to R_j \in \mathcal{B}$ such that there exists some $k \geq j$ such that $R_i \sim R_k \in \mathcal{B}$. But then by our construction of G, we would then have a cycle $\{(R_i, R_k), (R_k, R_i)\} \subseteq E$. And hence our algorithm will return "INCONSISTENT" as we have a back edge. Note, this fact that a cyclic directed graph has a cycle if and only if there is a back edge in its depth first search tree was discussed in lecture in Week 9 regarding topological sorting. If no such inconsistency exists, then G will have no such cycle. And hence we will return "CONSISTENT". Hence, our algorithm is correct.

Note, in effect our algorithm relies on the key ideas of topological sorting. However, we decided to explicitly write our algorithm in terms of back edges, as opposed to explicitly using topological sort since there may not be a topological sorting if our graph G has cycles.

Explanation of Runtime: Our algorithm is dominated by the depth first search. Hence, our algorithm takes as long as the depth first search which we know is $\mathcal{O}(n+m)$, where $n = \mathcal{O}(|V|)$ and $m = \mathcal{O}(|E|)$, as required.

Part c)

Consider the following algorithm. Note, we will assume as a precondition that $\mathcal{B}$ is consistent. Hence, our graph G as described in part a) will not have any cycles.

PotentialWinners($\mathcal{B}$)

```
1: if B only contains observations of the form R_i ~ R_j then
2:     return {R_1, R_2, ..., R_n}
3: end if
4: Construct the graph G = (V, E) as described in part a).
5: Run DFS (depth first search) on G and construct the DFS tree T.
6: Let π[v] be the parent of each vertex v ∈ V in our DFS tree T. Note, π[v] is Null if v
   has no parent node in T.
7:
8: losers ← {}
9: for each vertex v ∈ V do
10:    if π[v] is not Null in our DFS tree T then
11:        Add v to losers
12:    end if
13: end for
14:
15: potential ← {R_1, R_2, ..., R_n} \ losers
16: return potential
```

Explanation of Correctness: If $\mathcal{B}$ only contains observations of the form $R_i \sim R_j$, then every runner can potentially win, and hence our algorithm correctly returns $\{R_1, R_2, ..., R_n\}$. Otherwise, our algorithm runs DFS and then considers each vertex $v \in V$. If $\pi[v]$ is not Null, where π indicates parenthood in the DFS tree T, then we know that v cannot possibly win based on the construction of our graph G. i.e. Assume $v = R_j$ for some $j \in \{1, ..., n\}$ and $\pi[v]$ is not Null. Then in our graph we must have some R_i such that $(R_i, R_j) \in E$. Hence, we have some $R_i \to R_j \in \mathcal{B}$, and hence R_j cannot win as R_i finishes before R_j even begins. So R_j is a loser. Our algorithm then adds all such losers to the set $losers$. Finally, we return $potential = \{R_1, ..., R_n\} \setminus losers$ which are all the potential winners.

Explanation of Runtime: Our algorithm is dominated by the depth first search. Hence, our algorithm takes as long as the depth first search which we know is $\mathcal{O}(n+m)$, where $n = \mathcal{O}(|V|)$ and $m = \mathcal{O}(|E|)$, as required.