Quiz 2 Solutions

6.046 Spring 2013 Quiz 2
Histogram for Entire Quiz

Mean: 53.0; Median: 52.5; Standard deviation: 19.5.
INSTRUCTIONS
This take-home quiz contains 5 problems worth a total of 100 points. Your quiz solutions are due at 4:59pm on Monday, April 22, 2013 at 11:59pm on Wednesday, April 24, 2013. Late quizzes will not be accepted unless you obtain a Dean’s support or make prior arrangements with the course staff. You must submit your solutions electronically. The previous policy (from Quiz 1) about awarding a score of 30% to a blank answer does not apply to this quiz.

Guide to this quiz: For problems that ask you to design an efficient algorithm for a certain problem, your goal is to find the most efficient algorithm possible. Generally, the faster your algorithm, the more points you receive. For two asymptotically equal bounds, worst-case bounds are better than expected or amortized bounds. The best possible solution will receive full points if well written, but ample partial credit will be given for correct solutions, especially if they are well written. Bonus points may be awarded for exceptionally efficient or elegant solutions.

Plan your time wisely. Do not overwork, and get enough sleep. Your very first step should be to write up the most obvious algorithm for every problem, even if it is exponential time, and then work on improving your solutions, writing up each improved algorithm as you obtain it. In this way, at all times, you have a complete quiz that you could hand in.

Policy on academic honesty: The rules for this take-home quiz are like those for an in-class quiz, except that you may take the quiz home with you. As during an in-class quiz, you may not communicate with any person except members of the 6.046 staff about any aspect of the quiz, even if you have already handed in your quiz solutions. To communicate with the staff, send an email to 6046-staff@csail.mit.edu. We will not respond to your question if you send an email to the personal email of a staff member. In addition, you may not discuss any aspect of the quiz with anyone except the course staff until April 30, 2013. Note that this date is after the due date of the quiz.

This take-home quiz is “limited open book.” You may use your course notes, the CLRS textbook, and any of the materials posted on the course web page, but no other sources whatsoever may be consulted. For example, you may not use notes or solutions to problem sets, exams, etc. from other times that this course or other related courses have been taught. You may not use any materials on the World-Wide Web, including OCW. You probably won’t find information in these other sources that will help directly with these problems, but you may not use them regardless.

If at any time you feel that you may have violated this policy, it is imperative that you contact the course staff immediately. If you have any questions about what resources may or may not be used during the quiz, please send email to 6046-staff@csail.mit.edu.

Write-ups: Type your answers up on separate files (the templates will be given), and submit all your answers electronically, as you would do in the problem set. We strongly encourage you to submit your problems typed in LaTeX, since this makes it easier for us to read your solutions and understand them.

Your solutions are due, electronically, at 4:59pm on Monday the 22nd of April at 11:59pm on Wednesday the 24th of April. There is no hardcopy submission option for this test.
Your write-up for a problem should start with a topic paragraph that provides an executive summary of your solution. This executive summary should describe the problem you are solving, the techniques you use to solve it, any important assumptions you make, and the asymptotic bounds on the running time your algorithm achieves, including whether they are worst-case, expected, or amortized.

Write your solutions cleanly and concisely to maximize the chance that we understand them. When describing an algorithm, give an English description of the main idea of the algorithm. Adopt suitable notation. Use pseudocode if necessary to clarify your solution. Give examples, draw figures, and state invariants. A long-winded description of an algorithm’s execution should not replace a succinct description of the algorithm itself. Points will be taken off for overly convoluted solutions to the problems.

Provide short and convincing arguments for the correctness of your solutions. Do not regurgitate material presented in class. Cite algorithms and theorems from CLRS, lecture, and recitation to simplify your solutions. Do not waste effort proving facts that can simply be cited.

Be explicit about running time and algorithms. For example, don’t just say that you sort $n$ numbers, state that you are using MERGE-SORT, which sorts the $n$ numbers in $O(n \lg n)$ time in the worst case. If the problem contains multiple variables, analyze your algorithm in terms of all the variables, to the extent possible.

Part of the goal of this quiz is to test your engineering common sense. If you find that a question is unclear or ambiguous, make reasonable assumptions in order to solve the problem, and state clearly in your write-up what assumptions you have made. Be careful what you assume, however, because you will receive little credit if you make a strong assumption that renders a problem trivial.

**Bugs:** If you think that you’ve found a bug, send email to 6046-staff@csail.mit.edu. Corrections and clarifications will be sent to the class via email and posted on the class website. Check your email and the class website daily to avoid missing potentially important announcements.

**Good Luck!**
Problem 1. First Draft

Good news! You’ve recently been hired as the general manager of the new 6.046 football team, the Arizona Ordinals. Coming up soon is the draft, where you pick which players will be on your team, and you need to come up with a strategy to make the best team possible. Specifically:

1. There are $m$ positions you need to fill on your team, labeled 1 through $m$. (You may assume that all positions are distinct.)

2. There are $n \geq m$ players available in the draft. Each player $i$ has a skill level $s(i)$, and is capable of playing in some set of positions $P(i) \subseteq \{1, 2, \ldots, m\}$. Note that the skill depends only on the player and not on the position they are playing.

3. Your goal is to build a team $T$ of exactly $m$ players. Formally, a team consists of a set $S$ of players as well as an assignment $A : S \rightarrow \{1, 2, \ldots, m\}$ of selected players to positions they are capable of playing. That is, $A(i) = j$ means that player $i$ is assigned to position $j$, which must be $\in P(i)$.

4. The quality of a team is the sum of the skill levels of its players. That is, $q(S, A) = \sum_{i \in S} s(i)$.

Design an algorithm to make the best possible team (list of players as well as assigned positions), or output that no complete team exists. Be sure to analyze the running time in terms of both $n$ and $m$, and to compare running times, treat the relative size of the variables as $n \gg m$ (there are way more players in the draft than spots on your team).
Mean: 9.8; Median: 11.0; Standard deviation: 4.1.

**Solution:**  
**Exhaustive Solution:** Loop over all \( \binom{n}{m} \) sets of possible players, then loop over all \( m! \) assignments of these players to positions. If the assignment is valid (i.e. it assigns each player to a position they are capable of playing), then compute the quality of this team. Output the argmax over all possible teams. If no teams are feasible, output that no complete team exists. This takes time \( O(n^m) \) to loop over all possible teams, and time \( O(m) \) to check whether or not the team is feasible and compute its quality. So the total running time is \( O(mn^m) \).

**Max-Weight Matching Solution:** Make a bipartite graph with players on the left and positions on the right. Put an edge of weight \( s(i) \) between player \( i \) and position \( j \) if player \( i \) is capable of playing position \( j \), and none otherwise. Then every feasible team corresponds to a complete matching (by complete we mean matching of size \( m \)) in this graph. The team \( T = (S, A) \) corresponds to the matching that matches every player in \( i \) to the position \( A(i) \). Similarly, every complete matching in this graph corresponds to a team. If player \( i \) is matched to position \( j \) in the graph, then put \( i \) in \( S \) and set \( A(i) = j \). Finally, the weight of the matching is exactly the quality of the corresponding team. So our algorithm just builds this graph and finds the max-weight complete matching, if one
exists. If the algorithm outputs that none exists, then we output that no complete team exists. If the algorithm outputs a complete matching, then we know it corresponds to the optimal complete team.

We can build this graph in time $O(nm)$, and find the max-weight matching in time $O(n^2m + n^2 \log n)$ using an algorithm cited in class (Lecture 15). Recall that the runtime of this algorithm on a generic bipartite graph was $O(VE + V^2 \log V)$. For the graph we built, $V = O(n)$ and $E = O(nm)$, which yields $O(n^2m + n^2 \log n)$ when substituted in.

Common mistakes:

1. Using a suboptimal matching algorithm. Either forgetting to use Johnson’s trick as a black box (presented in lecture) or forgetting that the graph is bipartite and therefore we can use a better algorithm.

2. Forgetting to prove that the max-weight matching will always be complete, or augmenting your algorithm to find the max-weight complete matching (and not just a max-weight matching), or at least give some sentence addressing this issue.


**Max-Weight Matching with Improved Runtime:** A few students observed that you can augment this solution by first removing all but $m^2$ players from the start. For each position, we claim that the best team fills it with one of the top $m$ players for that position. This is because at most $m - 1$ of these players could possibly be filling other positions, so at least one of them is always available. That means that if a player outside the top $m$ is currently playing this position, we can always replace him with an unused player inside the top $m$.

So let’s first discard any player who is not one of the top $m$ players for any position. For each position, we can find the top $m$ players in time $O(n)$ using the rank finding algorithm presented in lecture (i.e. median finding but for the $m^{th}$ best player). So we can find the $O(m^2)$ players that we need to keep in time $O(nm)$. Afterwards, we build a bipartite graph with $O(m^2)$ nodes and $O(m^2)$ edges and run the same max-weight matching algorithm. The new runtime of the max-weight matching algorithm will just be $O(m^4 \log m)$. So the total runtime will be $O(nm + m^4 \log m)$.

**Further Improvement to Max-Weight Matching:** Lastly, with a good understanding of the nuts and bolts of the actual algorithm used to find the max-weight complete matching, we can improve the runtime to $O(nm^2 + nm \log n)$, or $O(nm + m^3 \log n)$. The algorithm will be almost exactly the one discussed in lecture, but we will just analyze the runtime more carefully for this special case. Recall that the algorithm starts with the null matching and iteratively finds augmenting paths until no more augmenting paths can be found. The naive runtime of this algorithm is equal to the number of times we need to find an augmenting path times the time it takes to find an augmenting path. For this instance, we can find up to $m$ augmenting paths before we have a complete matching. Using Bellman-Ford, we can find each augmenting path in time $O(VE)$. So the runtime of this algorithm would be $O(mVE)$. We would like to improve the runtime using Johnson’s trick, which was mentioned in class. Because we are no longer using a result from class as a black box, we should describe formally how to use Johnson’s approach, because it is not straightforward (actually, it’s
Recall that at each step, our residual graph contains an edge of weight 0 from a source to every unmatched player. An edge of weight $-S(p)$ from each player to every position they can play that they aren’t currently matched to, an edge of weight $S(p)$ from position $q$ to player $p$ if $p$ is currently assigned to position $q$, and an edge of weight 0 from every unfilled position to a sink.

Recall one last time: our plan is to use Dijkstra’s algorithm on a modified residual graph instead of Bellman-Ford on the actual residual graph to find a good s-t path. Let’s now say explicitly what potential functions we will use to augment the weights. Recall that Johnson’s approach stores a potential at every node $h(v)$ and augments the weight of every edge to be $w'(u, v) = w(u, v) + h(u) - h(v)$. The goal is to have potential so that $w'(u, v) \geq 0$ for all $u, v$ so that we can run Dijkstra’s algorithm instead of Bellman-Ford. We will only modify edges between players and positions, and have all edges from $s$ to players or positions to $t$ remain of weight 0. Additionally, we will require a stronger property that $w'(u, v) = 0$ for all backwards edges (i.e. for all edges that are currently in our matching and therefore go from positions to players).

Just to be absolutely clear, we want our potential function $h$ to satisfy the following properties:

1. For all players $p$ and positions $q$ such that $p$ is not currently assigned to $q$: $-S(p) + h(p) - h(q) \geq 0$.
2. For all players $p$ and positions $q$ such that $p$ is currently assigned to $q$: $S(p) + h(q) - h(p) \geq 0$.

We start with the following potentials:

1. Assign $h(p) = 0$ for all players $p$.
2. Assign $h(q) = \min_{p \mid q \in P(p)} \{-S(p)\}$ (i.e. negative of the maximum skill of any player who can play position $q$).

Clearly, these potentials can be computed in time $O(E)$. It is also clear that these potentials satisfy the desired properties. There are no edges from positions to players because we currently have no matching. For all edges from players to positions, we have $w(p, q) = -S(p), h(p) = 0, h(q) \leq -S(p)$, so $w(p, q) + h(p) - h(q) \geq 0$. So using this potentials, we can run Dijkstra’s algorithm to find the shortest path in this modified graph from $s$ to $t$. This corresponds to an augmenting path. Then we will update the matching according to this augmenting path and continue.

Next, we have to describe how to update the potentials efficiently from round to round.

1. Assume that currently we are using potential function $h$, and that $h$ is valid for the existing residual graph.
2. Let $d(v)$ be the cost of the shortest path from $s$ to $v$ in the residual graph using weights $w'$ (i.e. modified by the original potential $h$). Then set $h'(v) = d(v) + h(v)$ as the new potential of $v$.

It is again clear that we can update these potentials in the time it takes to run one iteration of Dijkstra. So we just need to argue that these new potentials are valid after we have augmented our maching with the shortest s-t path.
Let’s first consider \((p, q)\) edges that were previously matched (i.e. backwards in the original residual graph). Because \(p\) was matched, the only edge going into \(p\) in the residual graph is from \(q\). Therefore, \(d(p) = d(q) + S(p) - h(p) + h(q)\). If \((p, q)\) is still matched, then we want \(S(p) + h'(q) - h'(p) \geq 0\). Indeed, we have (using the fact that \(d(p) = d(q) + S(p) - h(p) + h(q)\)):

\[
S(p) + h'(q) - h'(p) = S(p) + h(q) + d(q) - h(p) - d(p) = S(p) + h(q) - h(p) - S(p) + h(p) - h(q) = 0.
\]

If \((p, q)\) is now unmatched, then instead we want \(-S(p) - h'(q) + h'(p) \geq 0\). Again using our recent observation, we have:

\[
-S(p) - h'(q) + h'(p) = -S(p) + h(p) + d(p) - h(q) - d(q) = -S(p) + h(p) - h(q) + S(p) - h(p) + h(q) = 0.
\]

Let’s now consider edges \((p, q)\) that were not matched previously, but will become matched now. Such edges necessarily lie along the shortest path from \(s\) to \(t\), as this is how we augment our matching. Therefore, we necessarily have \(d(q) = d(p) - S(p) + h(p) - h(q)\). As \((p, q)\) is now a backwards edge in our residual graph, we need to have \(S(p) + h'(q) - h'(p) \geq 0\). Indeed, we can expand this out to get:

\[
S(p) + h'(q) - h'(p) = S(p) + h(q) + d(q) - h(p) - d(p) = S(p) + h(q) - h(p) - S(p) + h(p) - h(q) = 0.
\]

Finally, let’s consider edges \((p, q)\) that were unmatched previously, and remain unmatched. For such edges, we know that \(-S(p) + h(p) - h(q) \geq 0\). Furthermore, because there is an edge of weight \(-S(p) + h(p) - h(q)\) from \(p\) to \(q\), we know that \(d(q) \leq d(p) - S(p) + h(p) - h(q)\) (also written as \(d(p) - d(q) - S(p) + h(p) - h(q) \geq 0\)). And now we want to show that \(-S(p) + h'(p) - h'(q) \geq 0\). Indeed, we have:

\[
-S(p) + h'(p) - h'(q) = -S(p) + h(p) + d(p) - h(q) - d(q) \geq -S(p) + h(p) - h(q) + d(p) - d(q) \geq 0
\]

So these new potentials are valid for the updated residual graph. So we can now run through our algorithm finding the shortest s-t path in the each residual graph and updating the potentials as we go, always maintaining a valid potential. It is obvious that after \(m\) iterations we will have a matching of size \(m\). The last step is showing that this is necessarily the max-weight complete matching. Assume for contradiction that there is a complete matching of larger weight. Then there is necessarily an augmenting cycle for our matching of positive weight, which corresponds to a negative-weight cycle in the residual graph when we finish. However, we know that there exists a valid potential function \(h\) when we finish, such that each \(w'(u, v) \geq 0\). So the modified weight of such a cycle necessarily satisfies:
$$0 \leq \sum_{e \in C} w'(e) = \sum_{e = (x,y) \in C} h(x) - h(y) + w(e) = \sum_{e \in C} w(e)$$

In other words, the existence of a valid potential function \( h \) guarantees that there are no negative cycles in the residual graph. If there are no negative cycles in the residual graph, there cannot be any positive-weight augmenting cycles in the matching we output. Therefore, the matching we output is optimal for its cardinality.

In conclusion, we described a complete algorithm that makes use of Johnson’s trick from all-pairs shortest-paths. We gave an algorithm to initialize and update the potentials at every step in the time it takes to run one iteration of Dijkstra’s algorithm. The total number of iterations we have to run is \( m \), and the runtime of each iteration is just one iteration of Dijkstra, which is \( O(E + V \log V) \). Plugging in to the basic matching algorithm, the runtime is \( O(nm^2 + nm \log n) \). Plugging this into the improved algorithm of the previous solution gives a runtime of \( O(nm + m^3 \log m) \).

**Greedy Solution:** Sort the players in decreasing order of skill. To simplify notation, we will denote by player \( i \) the player with the \( i \)th highest skill for the rest of the solution. We call a subset \( S \) of players (not necessarily a complete team) feasible if there exists an assignment \( A \) of players in \( S \) to \( |S| \) different positions, where player \( i \) is capable of playing position \( A(i) \) for all \( i \). Here is an overview of the algorithm we want to implement. The idea is that if a set of players is feasible, then every player contributes something to the team, because it’s possible to assign each player to a distinct position. If a set of players is infeasible, then there’s no way to put all the players on the field at the same time, and at least one player is wasted. So a reasonable approach is to process the players in order of skill, drafting anyone who keeps your current team feasible, and rejecting anyone who makes your team infeasible. Below is pseudocode for this approach:

1. Initialize \( S = \emptyset \)
2. for \( i = 1 \) to \( n \)
3. if \( S \cup \{i\} \) is feasible
4. Add \( i \) to \( S \).
5. return \( S \)

Two steps are missing to turn this “overview” into an actual algorithm. The missing steps are 1), we need to describe how to check if a set \( S \) is feasible and 2), we need to output the allocation used, not just the players. To make both changes at once, we can update the algorithm to store the following information. Let \( G \) be a graph with players in \( S \) on the left and positions on the right, with an edge (no weights) between \( i \) and \( j \) iff player \( i \) is capable of playing position \( j \). Let \( M \) be matching of size \( |S| \) in \( G \). Such a matching is guaranteed to exist because \( S \) is always feasible.
1. Initialize $S = \emptyset$, $G = \emptyset$, $M = \emptyset$.
2. for $i = 1$ to $n$
   3. Set $S' = S \cup \{i\}$.
   4. Set $G' = G$.
   5. Add $i$ as a left node to $G'$.
   6. Add an edge between $i$ and every position that $i$ can play in $G'$.
   7. if There is an augmenting path to $M$ in $G'$
      8. Update $S = S'$, $G = G'$, and augment $M$ using this augmenting path.
   9. else Continue.
10. return $(S, M)$

It is now clear that we output a complete team, including position assignments. We have also given explicitly an algorithm to determine whether or not $S'$ is feasible. So our complete algorithm would be to first sort the players by skill, then run the above algorithm, then output $(S, M)$ if $|S| = m$, and that no feasible solution exists otherwise. We now have to show that our algorithm is correct. That is, that the set $S$ output by the simplified algorithm is the best possible set of players (if one exists).

Assume for contradiction that there is some set of players $S'$ such that $\sum_{i \in S'} s(i) \geq \sum_{i \in S} s(i)$. Then there necessarily exists some $i^*$ such that the $i^*$-best player in $S'$ is better than the $i^*$-best player in $S$ (this is also true if $S'$ is a complete team but $S$ has fewer than $m$ players). Consider now $S_*$ and $S'_*$, which are $S$ and $S'$ after removing all players with skill below that of the $i^*$-best player in $S'$. Then clearly $|S'_*| > |S_*|$. As $S'_*$ is feasible, there is some matching $M'$ of players in $S'_*$ to positions they can play. Let $M$ be the corresponding matching for $S_*$, and consider the graph $M \cup M'$ (i.e. put edges from both $M$ and $M'$ in the graph).

As both $M$ and $M'$ are matchings, every node had degree at most one. Now that we put them together, every node has degree at most two. And every node with degree two is adjacent to exactly one edge from $M$ and exactly one edge from $M'$. Therefore, $M \cup M'$ is a disjoint union of cycles and simple paths. It is clear that there must be at least one simple path: because $|S'_*| > |S_*|$, there is at least one node on the left with only one adjacent edge. We also claim now that if we can find a simple path of odd length with one endpoint $x$ in $S'_*$, then we have found a contradiction. If such a path exists, then it is clearly an augmenting path for $M$: start from player $x$ and follow edges in $M'$ over to positions. Because $M'$ is a matching, if there is an edge back to the players, it is necessarily from $M$. So at the end of this path we will have taken only edges in $M'$ to get from players to positions, and only edges in $M$ to get from positions back to players. So this is an augmenting path for $M$. Because it has odd length, it increases the size of $|M|$ by one. Because $x \in S'_*$, we definitely processed $x$ while our current feasible set was $S_*$ (or possibly a subset of $S_*$). This argument shows that we can add $x$ to $S_*$ and still get a feasible set. So this contradicts the behavior of our algorithm.

Finally, we just need to show that a simple path of odd length with one endpoint in $S'_*$ necessarily exists. Let $X'$ be the players in $S'_*$ but not $S_*$. Let $X$ be the players in $S_*$ but not $S'_*$. Then every player in either $X$ or $X'$ is the endpoint of a simple path, and not contained in a cycle (because
they all have edges from only one matching, not both). Every path of even length necessarily has one endpoint in $X$ and the other in $X'$. This is because every path necessarily takes edges in $M'$ to get from players to positions, and edges in $M$ to get back (or vice versa). So if an even path starts at some $x \in X'$, the last edge it takes is in $M$, and therefore it ends at some $y \in X$. Therefore, it is not possible for every element of $X'$ to be in a path of even length, as $|X'| > |X|$, and every path of even length claims an element of $X$. So there is at least one path of odd length with one endpoint in $S'_*$, and we have found our contradiction.

To summarize, we first proved a technical claim about matchings: if $M$ and $M'$ are two matchings in the same graph, and $|M| < |M'|$, then $M$ can be augmented using edges in $M'$ to add a new node that is matched in $M'$ but not in $M$. We then observed that if $S'$ is better than $S$, that there is some cutoff $s(i^*)$ where $S'$ has more players of skill above $s(i^*)$ than $S$. So we considered the matchings $M'$ and $M$ induced by $S'_*$ and $S_*$ using only these players. Our technical claim shows that there is necessarily a way to augment $M$ to match an element $x$ of $S'_*$ without unmatching anyone in $S_*$. That means that our algorithm should have accepted $x$ as long as its current set was a subset of $S_*$. Because our algorithm processes players in decreasing order of skill, we definitely process $x$ while our current set is a subset of $S_*$, and therefore the algorithm should have accepted it, but didn’t, which gives us our contradiction. Therefore, we have shown that if a complete team exists, our algorithm will necessarily find one, and also that our algorithm finds the best feasible complete team.

We can compute the runtime as follows. There are $n$ iterations of the for loop. In each iteration, we need to find whether or not an augmenting path exists in a graph with at most $m^2$ edges (because $|S| \leq m$ always). This can be done using Ford-Fulkerson in time $O(m^2)$. Therefore, the total running time of the algorithm (including sorting the players at the beginning) is $O(n \log n + m^2 n)$.

**Greedy Solution with Improved Runtime:** This is basically the same algorithm as the greedy solution, with a modification to improve the runtime. In other words, we will just implement an improved version of the following algorithm from the greedy solution:

1. Initialize $S = \emptyset$
2. for $i = 1$ to $n$
   3. if $S \cup \{i\}$ is feasible
   4. Add $i$ to $S$.
5. return $S$

We already showed in the previous solution that any correct implementation of this algorithm is correct. The idea for improving runtime is the following: in the previous implementation, we checked to see if there was an augmenting path at each iteration from scratch. Checking for the existence of an augmenting path requires $O(m^2)$ work, which led to a runtime of $O(n m^2)$. Rather than checking for augmenting paths from scratch at each iteration, we will find a clever way to store and reuse work from previous iterations.

Specifically, at every iteration we will still store a bipartite graph $G = L \sqcup R$ where $L$ is the set of players that have already been drafted (i.e. $L = S$) and $R$ is the set of all positions. $G$ has an
edge between a player in $L$ and a position in $R$ if and only if that player is capable of playing that position. We also store a matching $M$ of size $|L|$ in $G$, which is an assignment of all players in $L$ to positions they are capable of playing. Lastly, we define the set $R^*(M) \subseteq R$ as follows: start with $R^*(M)$ to be all positions that are unmatched in $M$. Then, add to $R^*(M)$ all elements of $R$ that have an augmenting path (which is necessarily of even length because both endpoints are in $R$) to an element in $R^*(M)$. Notice that $R^*(M)$ is exactly the set of positions with the following property: $S \cup \{i\}$ is feasible if and only if $i$ is capable of playing a position in $R^*(M)$. This is because $S \cup \{i\}$ is feasible if and only if there is an augmenting path of odd length starting with $i$ to $M$ in $G$. If the first edge in such a path is from $i$ to some position $p$, then there is clearly an even length augmenting path from $p$ to some unmatched position, and therefore $p \in R^*(M)$.

So, assuming that we can maintain $R^*(M)$ throughout our algorithm, we can check whether or not $S \cup \{i\}$ is feasible in time $O(m)$ by just checking if $i$ can play any positions in $R^*(M)$. So the total runtime of our algorithm is now $O(nm)$ plus the total time it takes to update $R^*(M)$ across all iterations. It is easy to see that if we do not find an augmenting path during an iteration, that $R^*(M)$ does not change, and therefore no work is done updating $R^*(M)$ during such an iteration. If instead an augmenting path is found, then we have to remove the newly filled position from the set of unmatched positions, and recompute $R^*(M)$. This can be done via a DFS in a directed graph with at most $m^2$ edges, and therefore takes time $O(m^2)$. It is also clear that an augmenting path can be found at most $m$ times (as at most $m$ players can possibly be drafted). Therefore the total cost of maintaining $R^*(M)$ is $O(m^3)$. Therefore, the complete runtime of this implementation of the greedy algorithm (including sorting the players in the beginning) is $O(n \log n + nm + m^3)$. 

Problem 2. Mt. Cartesian Rescue Team

You are in charge of the rescue team at Mt. Cartesian, a popular ski resort located at scenic Quadrant I of the coordinate plane. You have received reports of \( n \) injured skiers at various locations on the mountain, and you need to rescue them as quickly as possible.

Each skier \( i \) is located at coordinates \((x_i, y_i)\), where \( i \in \{1, 2, \ldots, n\} \) and \( x_i, y_i > 0 \). Given this information, you must dispatch a number of rescue snowmobiles from the summit of the mountain, which is located at \((0, 0)\), to collectively rescue all \( n \) skiers by locating them, picking them up, and dropping them off at a medical center located at \((x^*, y^*)\), where \( x^* > x_i \) and \( y^* > y_i \) for all \( i \). Any one snowmobile can accommodate as many skiers as necessary. However, due to strong wind conditions, a snowmobile may only move such that its \( x \) coordinate never decreases and its \( y \) coordinate never decreases. In other words, once a snowmobile picks up a skier located at \((x_i, y_i)\), then any other skier \((x_j, y_j)\) that it picks up later must satisfy \( x_j \geq x_i \) and \( y_j \geq y_i \).

Design an algorithm that, given \((x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\), determines the minimum number of snowmobiles that need to be dispatched and how to dispatch them (i.e., specify each snowmobile’s rescue route) so that they rescue every skier.
Mean: 11.8; Median: 12.0; Standard deviation: 5.0.

Solution: There are several approaches to this problem. First we will present the best solution.

**Greedy solution.** Sort all the points by increasing $x$ value, then by increasing $y$ value to break ties. (For example, $(3, 4)$ comes before $(3, 5)$, which comes before $(4, 4)$.) For each point $(x_i, y_i)$ in this sorted list, find the snowmobile $k$ whose last point’s $y$-coordinate has the highest value less than or equal to $y_i$, and add $(x_i, y_i)$ to $k$’s path. If no such snowmobile exists, dispatch a new snowmobile to $(x_i, y_i)$.

To do this efficiently, keep track of the current snowmobiles and their paths as elements in a self-balancing binary search tree $B$, where the key of each element is the $y$-coordinate of the last point in the corresponding snowmobile’s path. Then finding the correct snowmobile reduces to finding the predecessor of $y$ in $B$, which can be done in $O(\lg n)$ time. Therefore the entire algorithm can run in $O(n \lg n)$ time.

Now we prove correctness. Let $S$ be the set of points given. Observe that the points in $S$ form a partially ordered set, where $(x_i, y_i) \preceq (x_j, y_j)$ if and only if $x_i \leq x_j$ and $y_i \leq y_j$. Define a path to be a sequence of distinct points $P_1, P_2, \ldots, P_k$ such that $P_1 \preceq P_2 \preceq \ldots \preceq P_k$. 

We start by showing that this algorithm always generates valid paths. This is very straightforward to show. A point \((x_j, y_j)\) can only get appended to a path ending with \((x_i, y_i)\) if \(x_i \leq x_j\) since the points are processed in order of increasing \(x\)-coordinate, and we require \(y_i \leq y_j\) to actually append it. Therefore every snowmobile will follow a valid path.

To prove that this algorithm is optimal, we show a relationship between paths that cover every skier and antichains in \(S\), that is, subsets \(A \subseteq S\) such that for any two distinct points \(a, b \in A\), \(a \not\leq b\) and \(b \not\leq a\).

We first claim that the minimum number of paths \(p_{\text{min}}\) required to cover all skiers is at least the length of the longest antichain in \(S\). Let \(m\) be the length of the longest antichain in \(S\), and assume for purposes of contradiction that \(S\) can be covered in fewer than \(m\) valid paths. Then by Pigeonhole at least two elements in that longest antichain must be in the same path, which cannot be the case. Therefore \(p_{\text{min}} \geq m\).

Now we show that the number of paths \(p\) our algorithm generates does not exceed \(m\) by generating an antichain of length \(p\). Number the paths from 1 through \(p\) in the order in which they were created. For every point \((x_i, y_i)\), let \(\pi(x_i, y_i)\) denote the number of the path that \((x_i, y_i)\) belongs to.

First, we need the following lemma:

**Lemma.** Let \(P_1, \ldots, P_k\) be the last points in the paths numbered 1, \ldots, \(k\), respectively, at any stage of the algorithm. Then the \(y\)-coordinates of \(P_1, \ldots, P_k\) are in strictly decreasing order.

**Proof.** By induction on the number of points added.

Base case: When there’s only one point, there is only one path, so this is trivially true.

Inductive step: Suppose this invariant holds after \(u\) points are added. When we add the \((u + 1)\)th point, there are two cases:

Case 1: \((x_{u+1}, y_{u+1})\) is added to an existing path. Suppose it is added to \(P_j\). Let \((a_j, b_j)\) be the last point in \(P_j\). Then we must have \(b_j \leq y_{u+1}\). If \(j = 1\), the invariant will obviously still hold, as we may increase the \(y\)-coordinate of the last point in \(P_j\) without any worries. If \(j > 1\), let \((a_{j-1}, b_{j-1})\) be the last point in \(P_{j-1}\). We know that \(b_{j-1} > y_{u+1}\) because otherwise we would have added \((x_{u+1}, y_{u+1})\) to \(P_{j-1}\) instead of \(P_j\). Therefore this invariant holds in this case.

Case 2: \((x_{u+1}, y_{u+1})\) is the beginning of a new path \(P_{k+1}\). This only happens when \(y_{u+1}\) is less than the \(y\)-coordinate of the last point in \(P_k\), so the invariant will still hold. \(\Box\)

With this invariant, we may now make the following claim:

**Claim.** Every point \((x_i, y_i)\) is part of an antichain \(A\) of length \(\pi(x_i, y_i)\) such that for every other point \((x_j, y_j) \in A, x_j < x_i\) and \(y_j > y_i\).

**Proof.** By strong induction on \(i\).

Base case: \(i = 1\). Clearly \(\pi(x_1, y_1) = 1\), and \(\{(x_1, y_1)\}\) is a fitting antichain.

Inductive step: Suppose this is the case for all \(i \in \{1, \ldots, u\}\). If \(\pi(x_{u+1}, y_{u+1}) = 1\), then \(\{(x_{u+1}, y_{u+1})\}\) is a fitting anti-chain. Otherwise, let \((x_j, y_j)\) be the point at the end of \(P_{\pi(x_{u+1}, y_{u+1})-1}\). Since it has already been accounted for, \(j \leq u\), and by the inductive hypothesis, it is part of an
antichain \( A \) of length \( \pi(x_j, y_j) = \pi(x_{u+1}, y_{u+1}) - 1 \). Then \( A' = A \cup \{(x_{u+1}, y_{u+1})\} \) is the desired antichain. By the invariant, \( y_j > y_{u+1} \), so in order to process point \( j \) before point \( u + 1 \), we must have \( x_j < x_{u+1} \). Furthermore, for any other point \( (x', y') \in A \), \( x' < x_j \) and \( y' > y_j \), so we must have \( x_j < x_{u+1} \) and \( y_j > y_{u+1} \). Therefore \( A' \) has the desired properties, and the induction is complete. \( \square \)

Since there are \( p \) paths, any point in \( P_p \) can be used to generate an antichain of length \( p \), so the maximum antichain length must be at least \( p \). Therefore \( m \geq p \). But we already showed that \( p_{\text{min}} \geq m \), so therefore \( p_{\text{min}} \geq p \) by transitivity. However, we must also have \( p_{\text{min}} \leq p \) by virtue of the fact that \( p_{\text{min}} \) is the best we can do. Therefore \( p_{\text{min}} = p \), which means that our greedy algorithm does indeed minimize the number of paths needed to cover all the points.

**Alternative greedy solution 1.** This algorithm still builds paths in the same greedy manner as above, but it only builds one path at a time. After sorting into a list \( L \), take the first point in \( L \) and add it to a new path \( p \). Iterate through the rest of \( L \), appending a point \((x, y)\) to \( p \) whenever it is legal to do so. Once you reach the last point in \( L \), remove all elements that are in \( p \) from \( L \) and repeat until \( L \) is empty. The proof of correctness is very similar, as the paths generated are the same as those of the optimal solution. If you implement \( L \) as a linked list, each pass will take \( O(n) \) time. Since there can be at most \( n \) paths in the worst case, this algorithm runs in \( O(n^2) \) time.

**Alternative greedy solution 2.** Similar to alternative solution 1, but instead of scanning through a linked list, put all the points into a 2-D range tree \( T \). To figure out the successor of a point \((x, y)\), simply run \( T.\text{Range-Query}([x, x^*], [y, y^*]) \) and pull out the first point that is strictly larger. Once again, you build one path at a time and remove points from \( T \) once they’re added to the path. Since you only call \( \text{Range-Query} \) on each point \( O(1) \) times, this algorithm will run in \( O(n \lg^2 n) \) time.

**Maximum bipartite matching solution.** Create a bipartite graph consisting of \( 2n \) nodes labeled \( a_1, \ldots, a_n, b_1, \ldots, b_n \). For all \( i, j \) such that \( 1 \leq i, j \leq n \), draw an edge from \( a_i \) to \( b_j \) if and only if a snowmobile can legally travel from \((x_i, y_i)\) to \((x_j, y_j)\), i.e., if \( x_j \geq x_i \) and \( y_j \geq y_i \). Run your favorite maximum bipartite matching algorithm on this graph. The resulting matching links each skier \( i \) to the next skier \( j \) in \( i \)'s path, or denotes that \( i \) is the last skier in his/her path if \( a_i \) does not have a match.

This will work because if \( m \) is the size of a matching of the graph above, then there will be \( n - m \) skiers who will be at the end of a path, implying that there exists a solution involving \( n - m \) paths. To minimize \( n - m \), we must maximize \( m \).

The graph we formed has \( O(n) \) vertices and can have up to \( O(n^2) \) edges. Therefore, depending on which matching algorithm you use, you can get running times of \( O(VE) = O(n^3) \), \( O(V^{1/2}E) = O(n^{2.5}) \), or \( O(V^{2.376}) = O(n^{2.376}) \).

**Exhaustive solution.** Look at all \( n! \) permutations of the \( n \) skiers. For each permutation \( \pi \), figure out how many snowmobiles would be required to pick up the skiers in the exact order specified by \( \pi \). Then choose the one that yields the fewest number of snowmobiles. It takes \( O(n) \) time to scan each permutation, so the total running time is \( O(n \cdot n!) \).

There were a couple of common mistakes among students’ solutions:
1. Some students attempted a different greedy approach of pulling out the longest possible path. Unfortunately, this won't always work, as shown by this counterexample:

There is a single possible path of length 4, but it forces the remaining two points to be singletons, yielding three paths. However, it is possible to cover all these points with two paths:

2. Some students stated that the minimum number of paths needed equals the number of points who do not have any legal successors. The number of such points is a lower bound for the minimum number of paths, but these two quantities are not always equal. For example, in a $k \times k$ lattice of points, only one point (namely, the top-right one) has no legal successors, but the lattice requires $k$ paths to cover all points.

3. Some students tried to sort the points by Euclidean distance or by Manhattan distance and then proceeded with the first alternative greedy solution (building one path at a time). Un-
fortunately, this algorithm fails on the input \([(2, 11), (3, 10), (4, 12), (50, 10.5), (51, 9.5)]\) because it will put \((3, 10)\) and \((4, 12)\) in the same path, leaving the remaining points as singletons. However, the optimal solution here is 3 paths: \([(2, 11), (4, 12)]\), \([(3, 10), (50, 10.5)]\), and \([(51, 9.5)]\).
Problem 3. Casting Reality

You are the casting production assistant for the upcoming reality TV show, *Algorithm Idol: Project Runtime*. You have measured the algorithmic skill level of each applicant $i \in \{1, 2, \ldots, n\}$ as a positive real number $s(i) \in \mathbb{R}^+$. The producer of the show, Cosrik Daskamaine, wants you to select the participants for the show according to the following criterion: the skill level of no single participant is greater than the sum of the skill levels of any other $d$ distinct participants. (This policy is meant to make the show more entertaining by motivating any group of $d$ participants to gang up against a stronger participant.) You may assume that $d$ is a parameter much smaller than $n$.

The producer wants the show to run for as long as possible, so he would like to select as many applicants as possible. Design an algorithm that, given the list of the applicants, their skill levels, and $d$, finds the largest set of applicants that satisfy the above criterion. Be sure to analyze the running time of your algorithm in terms of both $n$ and $d$. 


Solution:  Let us refer to the optimal set as $P$. Make an array out of the skills. Sort the array increasing by skill and call the sorted array $C$.

**Required Conditions on Desired Set**

Note that the following must be true about $P$:

- if $C[i] \in P$ and $C[j] \in P$, then for all $k \in \{j, \ldots, i\}$ we have $k \in P$. That is, the optimal set is a contiguous sub-sequence of the sorted set $C$,
- a candidate set is valid if and only if the $d$ weakest players sum to more than the best player.

**Pre-Computing Sums**

Let us now compute the sums of all contiguous $d$-length sub-sequences of $C$. We can do so in linear time (given the sorted list) by the following scheme:

- Compute the sum of $C[1 : d] = s(1)$ by adding up all the $d$ elements
For each $i \in \{d + 1, \ldots, c - d\}$: $s(i) = s(i - 1) + C[i + d - 1] - C[i].$

**Considering Candidate Sets**

Now we can start with a starting pointer to the smallest element and an ending pointer to the $(d + 1)^{st}$ smallest. By the notes above, each start/end index pair corresponds to a candidate set. For each starting index, we wish to know the largest ending index such that the balance requirement is not violated. This can be done by moving the ending pointer to the right and checking (in $O(1)$, since sums are pre-computed). Once we have found a final ending location for a given start index, we move the start index up by one, and do the same. However, note that the search for a ending index can start from the ending index of the previous start index. For each start index, save the length of the maximum allowed sub-sequence (denote $l(i)$ for starting indices $i$).

**Largest Set**

Now that we know the largest set possible from each starting index, we can just select the largest of all such sets. That is, the largest set size if $\max_i l(i)$ and the corresponding set is the one that starts at the index $i^*$ and includes the $l(i^*)$ applicants starting index $i^*$.

**Running Time**

This takes $O(n \log n)$ because we had to sort initially. Pre-computing the sums takes $O(n)$ and browsing for the largest set also takes $O(n)$.

**Common Alternative Solution 1**

Some people used a similar method but did a binary search in the sorted array of skill for the max possible skill that met the constraint for every set of $d$ applicants in the precomputed sums. This was also $O(n \log n)$ and received full points if written correctly.

**Common Alternative Solution 2**

Some people did not pre-compute the sums, and had an additional factor of $O(dn)$ in the solution.

**Common Alternative Solution 3**

Some people considered all $O(n^2)$ start and end index combinations, and had their running time be $O(n^2)$. 
Problem 4. Live Royalty

You are planning a museum exhibit about royal lineage in the modern day. You have modeled the entire royal family as a rooted tree with \( n \) nodes, and observe that its maximum degree (number of children) of the nodes is \( \Delta \). You have been assigned \( k \) walls in the museum to put up \( k \) panels. Aesthetically, you decide that each panel should show exactly one rooted subtree of the family tree, that is, all descendants of a particular node. (This makes it easy to title each panel.) Your exhibit focuses on royal family members that are still alive, which happen to all be leaves in the tree. Your goal is to display all of these leaves in the \( k \) panels, but to avoid clutter, you would like to minimize the number of leaves you display that are not alive.

Formally, design an algorithm that, given a rooted tree with \( n \) nodes of maximum degree \( \Delta \), a target set of leaves \( L \) (the royal descendants that are still alive today), and a number \( k \), finds \( k \) rooted subtrees of the tree that together contain all leaves in \( L \) and, subject to this constraint, minimize the number of leaves not in \( L \).

Be sure to analyze your running time in terms of \( n, \Delta, \) and \( k \). To compare running times, you should treat the relative size of the variables as \( n \gg k \gg \Delta \) (there are way more members of the royal family than room in your display, and there is way more room in your display than the maximum number of children had by any single royal family member).
Mean: 8.1; Median: 8.0; Standard deviation: 6.4.

Solution: Many students asked whether the subtrees need to be disjoint, or whether there had to be exactly \( k \) or at most \( k \) chosen subtrees. Assumptions either way on these questions had no significant impact on the algorithms. Here we assume that the solution can consist of \( \leq k \) subtrees, which allows us to assume that the optimal solution has no subtree contained in another. (Equivalently, we could allow individual leaves in the solution, but then we would need to allow descendant subtrees.)

Best Solution: Executive summary. The best solution found by (two) students actually beats the solution we had in mind. It runs in \( O(nk) \) time via a reduction to finding the minimum-weight \( s-t \) path that uses at most \( k \) edges, which can be solved by a modified Bellman-Ford Algorithm.

Best Solution: Details. First we perform a post-order traversal of the tree to count the number \( v.\text{live} \) of living descendant leaves and the number \( v.\text{dead} \) of dead descendant leaves of each node \( v \). At the leaf level, one of these numbers is 1 and the other is 0, as specified in the input by the status of the leaf. At each nonleaf node, we sum the values of the children. Because a tree has one fewer edge than it has nodes, this procedure takes \( O(n) \) time.
Next we construct a weighted directed graph $G$ by a depth-first search through all nodes $v$ in the tree with $v.\alive > 0$, i.e., with at least one living descendant leaf. The depth-first search visits the living leaves of the tree in some order $\ell_1, \ell_2, \ldots, \ell_k$ (dependent on the arbitrary order that we visit the children of each node). During the depth-first search, we form a linked list on these leaves, so that $\ell_i.\next = \ell_{i+1}$ and $\ell_i.\prev = \ell_{i-1}$. For every living leaf $\ell_i$ in the tree, we form a vertex $x_i$ in $G$. We also form a dummy vertex $x_0$ in $G$. For every node $v$ in the tree with descendant living leaves $\ell_i, \ell_{i+1}, \ldots, \ell_j$, we form a directed edge $(x_{i-1}, x_j)$ in $G$, of weight $v.\dead$. This graph $G$ has $O(n)$ vertices and edges.

Now we claim that the optimal solution to the original problem is given by the shortest $\leq k$-edge path from $x_0$ to $x_k$ in $G$. Every such path $x_0 \to x_{i_1} \to x_{i_2} \to \cdots \to x_{i_k-1} \to x_k$ can be mapped to the corresponding nodes $v_1, v_2, \ldots, v_k$ in the tree, where $v_j$ has descendant living leaves $x_{i_{j-1}+1}, x_{i_{j-1}+2}, \ldots, x_{i_j}$ (where $i_0 = 0$ and $i_k = k$). These $k$ nodes clearly “cover” all living leaves, so form a feasible solution to Live Royalty. The number of included dead descendant leaves is equal to the weight of the path (by definition of weight). On the other hand, any feasible solution to Live Royalty, consisting of nodes $v_1, v_2, \ldots, v_k$, can be converted to a $\leq k$-edge path as follows. If any $v_j$ is the ancestor of another $v_{j'}$, we remove the latter node from the solution. Then we can relabel the $v_j$s so that they appear in order from left to right in the tree. Suppose node $v_i$ has descendant living leaves $x_{a_i}, x_{a_i+1}, \ldots, x_{b_i}$. Because the solution must cover all living leaves, $a_i$ must equal 1, and $b_i$ must equal $k$, and $a_i+1 - 1$ for $1 \leq i < k$. Thus $x_0 \to x_{a_1} \to x_{b_2} \to \cdots \to x_{b_k}$ must form the desired path. Its weight equals the number of included dead descendant leaves in the solution $v_1, v_2, \ldots, v_k$.

Finally, we show how to find the shortest $k$-edge path from $s$ to $t$ in a weighted directed graph in $O((V + E)k)$ time, which in our case is $O(nk)$. The approach is to use $k$ iterations of a modified Bellman-Ford algorithm (perhaps easier seen as a shortest-paths dynamic program). Initially, we set $d_0[v] = \infty$ for all $v$, and then set $d_0[s] = 0$. In each iteration $i = 1, 2, \ldots, k$, we visit every vertex $v$, and set $d_i[v] = \min\{d_{i-1}[u] + w(u,v) \mid (u,v) \in E\}$, where $w(v,v) = 0$.

**Good Solution: Executive summary.** The best solution we had in mind (which still received full score) runs in $O(nk^2)$ time. The algorithm uses dynamic programming with $O(nk)$ subproblems and $O(k)$ time per subproblem, by careful dynamic programming at each node and use of tree bounds to remove a factor of $\Delta$.

**Good Solution: Details.** As in the previous solution, first we spend $O(n)$ time to count the number $v.\alive$ of living descendant leaves and the number $v.\dead$ of dead descendant leaves of each node $v$. We then remove all nodes $v$ with $v.\alive = 0$, as they do not affect the solution.

The dynamic program has $O(nk)$ subproblems, each defined by a nonleaf node $v$, a child index $i$, and a budget $b$. The goal of the subproblem is to find $b$ rooted subtrees, among the first $i$ children of $v$, that cover all living descendant leaves of those $i$ children, and subject to that condition, minimize the number of included dead descendant leaves. There are $\Delta k$ subproblems per node, so at first glance there may appear to be $O(nk\Delta)$ total, but if we instead charge each subproblem $(v,i,b)$ to the $i$th child of $v$, then we see that there are just $O(nk)$ subproblems.

To solve a subproblem $(v,i,b)$, we guess how much of the budget $b$ should be used for the $i$th child
u of v, by choosing $b' \in \{1, \ldots, b\}$. If $b' = 1$, we also guess whether $u$ is the root of one of the chosen subtrees. If yes, we add $u\text{.dead}$ to the solution of subproblem $(v, i - 1, b - b')$. If not, we sum the solutions of subproblems $(v, i - 1, b - b')$ and $(u, \text{degree}(u), b')$. Taking the minimum over all guesses produces the desired cost, and following parent pointers reveals the solution attaining this cost.

At the top level, we introduce an artificial root node $r'$ with a single child equal to the original root $r$ of the tree, and return the solution to $(r, 1, k)$. For base cases, if $i = 0$, we return 0; if $i > 0$ and $b = 0$, we return $\infty$ (as we have already removed nodes with no living descendant leaves).

Because there are $O(k)$ guesses, and we spend $O(1)$ time per guess, we spend $O(k)$ time for each of the $O(nk)$ subproblems, for a total running time of $O(nk^2)$. 

Problem 5. Sign Here

You’ve been hired to advise the amusement park *I Have Truly Found Paradise* on how to construct their “You Are Here” signs. The park has \( N \) rides and \( N \) footpaths connecting rides, so you can model the park as a connected, undirected graph \( G = (V, E) \) with \( |V| = |E| = N \). Each “You Are Here” sign contains a map (that is, a drawing of \( G \)) where exactly one vertex has been labeled with “You Are Here,” and all edges and all other vertices are unlabeled.

1. You find two “You Are Here” maps on the ground, but you don’t know where they’re from. (They might even be from two different amusement parks!) You want to determine whether the two maps might be from the same spot.

   More precisely, two maps \( G \) and \( H \) are *interchangeable* if there is a bijection \( f \) from vertices of \( G \) to vertices of \( H \) such that \( f \) maps the “You Are Here” vertex of \( G \) to the “You Are Here” vertex of \( H \), and such that vertices \( u \) and \( v \) of \( G \) are adjacent in \( G \) if and only if vertices \( f(u) \) and \( f(v) \) of \( H \) are adjacent in \( H \).

   Design an efficient algorithm for determining whether two maps are interchangeable.

2. *I Have Truly Found Paradise* wants to print \( N \) “You Are Here” signs, one to place at each ride. However, to save on printing costs, they would like to create as few distinct signs as possible. In particular, if multiple “You Are Here” maps are interchangeable with one another, they can print multiple copies of one sign. Design an efficient algorithm for counting how many distinct signs the amusement park needs to create.

For instance, consider the following amusement park maps with \( N = 4 \). A \( \circ \) represents an unlabeled ride, and a \( \bullet \) represents a ride labeled “You Are Here.”

```
  O
 /|
/ |
  O
```

This park would need only 3 distinct signs because the third and fourth maps are interchangeable. Your algorithm only needs to output the number “3”; you do not need to construct the actual graphs.
Mean: 7.3; Median: 7.5; Standard deviation: 5.4.

**Solution:** This problem is essentially graph isomorphism, which sounds hard because the general version is in NP (and likely not in P).

We immediately have a lower bound of $O(N)$ for both parts because it’s impossible to calculate an answer without considering the entire input graphs. Surprisingly, we can solve the first part in $O(N)$ worst-case and the second part in $O(N)$ expected.

**Part 1:** We can assume that $|V| = |E| = N$ for both graphs, otherwise we immediately know that they are not interchangeable.

**Brute force:** There are $(N - 1)!$ ways to correspond vertices of the two graphs (you know that the labeled vertices must correspond), and each correspondence takes $O(N)$ time to check that the edges match, so $O(N \cdot (N - 1)!)$ = $O(N!)$.

**Improved using properties of the graph:** Because graph isomorphism is hard, we need to take advantage of the fact that we’re solving a very limited case of this problem in order to reduce the runtime. If we have a connected graph with $N$ vertices and $N$ edges, then there must be exactly
one cycle. You can view the graph as a cycle with trees hanging off at each cycle vertex. The trees containing the labeled vertices of $G$ and $H$ must correspond, so there are only two ways to potentially correspond the cycles. For both of these two ways, go around the cycle checking if the corresponding trees hanging off the cycle are interchangeable.

Checking if corresponding trees are interchangeable/isomorphic is a much easier problem. Given two trees, first check that the degrees of their roots match. Call this degree $d$. For each of the $d^2$ pairs of one child from the first tree and one child from the second, recursively check if the two subtrees are isomorphic. The worst case is when all $d$ children have equal size, so if this tree has $n$ nodes, the recurrence is $T(n) = d^2 T((n-1)/d) + 1 < d^2 T(n/d) + 1$, which solves to $T(n) = n^2$ by the master theorem.

Don’t forget that you need to make sure the labeled vertices can correspond! If the two sets of children you’re considering both have a labeled vertex, those labeled vertices must correspond. If only one of those sets has a labeled vertex, the trees are not isomorphic. This gives an $O(N^2)$ solution to the original problem.

**Common mistake:** Many students incorrectly stated that you can match up $d$ children in $O(d)$ time by matching up pairs of children with the same degree or the same subtree size. Unfortunately, these strategies don’t work. Consider the example below: this strategy might match $A$ with $C$ and match $B$ with $D$, thereby concluding that the graphs are not interchangeable. Instead, if there are multiple children of equal degree/subtree size, you don’t know which correspond with each other, and picking an arbitrary correspondence isn’t going to work. Thus, you need to check all $d^2$ pairs in the worst case.

**Bottom-up approach, using hashing:** We can improve the time it takes to check that trees are isomorphic. Let’s ignore the labels temporarily and treat all vertices as unlabeled. Start at the bottom of the trees and, moving up, compute a hash value for each subtree. A subtree’s hash is based off the hashes of its children. However, we don’t want order to matter, so radix sort the child hashes first and then hash this entire sequence. It now takes $O(d)$ time to compute the hash for a node with $d$ children, so we can compute the hash of the root of a tree of size $n$ in time $O(n)$. To check if two subtrees are equivalent, just compare their hashes.
To solve the original problem, you merely need to go through the two potential ways to correspond the cycles and, for each one, see if each pair of corresponding trees have the same hash. Finally, verify that the subtrees rooted at both labeled vertices have the same hash. This improves the running time of the overall algorithm to $O(N)$.

**Alternate optimal solution, by converting to trees:** Another optimal solution involves first converting both $G$ and $H$ to trees. Find the vertex on $G$’s cycle that is closest to $G$’s “You Are Here” vertex. Arbitrarily remove one of the cycle edges adjacent to this vertex, forming a tree $G'$. We could repeat this same process for $H$ to yield $H'$, but because of the arbitrary choice between two edges, we lose the guarantee that $G$ and $H$ being interchangeable implies that $G'$ and $H'$ are.

Instead, form $H'_1$ and $H'_2$, as above but letting each tree correspond to one of the two arbitrary choices. Now $G$ and $H$ are interchangeable if and only if either $G$ and $H'_1$ are or $G$ and $H'_2$ are. This reduces to calling our tree isomorphism subroutine two times, which takes $O(N)$ time.

If you had a solution like this, you need to be very careful that the edges you remove do in fact correspond; many students messed this up.

**Part 2:** We assume that the input is a single graph with $|V| = |E| = N$ and no labeled vertex. (If you assumed that you were given $N$ graphs as input, we docked points because this blows up the input size.)

**Brute force:** Clearly, if we want the largest subset, all we have to do is check elements pairwise for isomorphism and, if they are isomorphic, delete one from the set. (This works because isomorphism is transitive.) There are $N^2$ pairs, so the running time is $O(N^2 \cdot T_{iso})$, where $T_{iso}$ is whatever running time you found in part 1.

**Improved using properties of the graph:** One reason we may have duplicate maps is that a rotation or reflection of the cycle yields an isomorphic graph. Say there are $C$ vertices in the cycle. If all trees coming off the cycle were non-interchangeable, we could place the “You Are Here” vertex in any of these $C$ trees. However, if some trees are interchangeable (as in the example given in the problem statement), we may have some interchangeable signs.

In particular, this happens when there are two cycle vertices $u$ and $v$ such that the cycle going clockwise or counterclockwise starting at $u$ looks the same as the cycle going clockwise or counterclockwise starting at $v$, where “looks the same” means that the same (or interchangeable) trees appear along the cycle in the same order. If this happens, then you can place the “You Are Here” vertex either in $u$’s tree or $v$’s tree, but not in both. One way to capture “looking the same” from a vertex $v$ is to go around the cycle starting at $v$, compute the hash of each tree you visit, and concatenate these hashes into a length-$C$ tuple. Now define a hash function that hashes these length-$C$ tuples into a hash table.

Our goal is to create a set $S$ of cycle vertices whose trees may contain the labeled vertex without creating interchangeable maps; initially, $S$ is the empty set. For each of the $C$ cycle vertices, create two length-$C$ strings (clockwise and counterclockwise from this vertex), hash these, and insert them into the hash table. If neither hash was in the table already, then add this vertex to $S$ (because it doesn’t look the same as anything we’ve seen so far). If only one hash was in the table already, also add this vertex to $S$ (as this must be a spurious collision, since an actual match would by
construction cause both hashes to already be in the table). If both hashes were in the table already, then do not add this vertex in $S$ (because we’ve already added to $S$ a vertex that “looks the same”). Well, not quite; we should first verify (via a manual comparison) that the tuples which mapped to these hashes actually match and that this wasn’t a spurious collision.

This correctly populates the set $S$. We created $O(C)$ hashes in $O(C)$ time each, and (assuming the table is large enough) we have an expected $O(1)$ collisions causing $O(C)$ extra work each. This is in total $O(C^2)$.

We know that the labeled vertex must be within a tree in $S$. So now, for each of the trees in $S$, we need to count how many non-isomorphic ways there are to place the vertex within the tree. We will use a bottom-up dynamic program that relies on having a hash for each subtree, as we calculated in part 1. We can start at the bottom of the tree and say that for each leaf, there is one way to pick the labeled vertex. For internal nodes, the number of ways is the sum of these computed values over all children with unique hash values. This takes $O(n)$ time overall to calculate, if there are $n$ vertices in the tree. Summing over all $C$ trees, this takes $O(N)$ at worst. Thus, the overall running time of this algorithm is $O(N^2)$.

**Using Rabin-Karp:** We can speed up the construction of set $S$ from $O(C^2)$ to $O(C)$ expected using a rolling hash, as taught in CLRS (see chapter 32.2 on Rabin-Karp) or in 6.006.

In short, what this gives us is: Once we compute our first hash in $O(C)$ time, we can compute each additional hash in $O(1)$ time by re-using the computation from the previous hash. This lets us compute all $O(C)$ hashes in $O(C)$ time total. Now the total running time is $O(C)$ for hashing, $O(C)$ expected for verifying collisions, and $O(N)$ for counting within the trees. This is $O(N)$ expected overall.

**Possibly even faster:** The bottleneck is now the $O(1)$ expected number of collisions. The Knuth-Morris-Pratt algorithm (see CLRS chapter 32.4) reduces the running time of string matching from $O(N)$ expected (as in Rabin-Karp) to $O(N)$ worst case, so modifying Knuth-Morris-Pratt to solve this problem would yield an $O(N)$ worst-case solution overall.

Another attempt at a linear time worst-case solution is: We only have a match if both the clockwise and counterclockwise hashes collide, and those two collisions are both with the same string (hashed in opposite directions). Try to find a rolling hash function $f$ with the special property that, if $f(a_1 \ldots a_k) = f(b_1 \ldots b_k)$, then $f(a_k \ldots a_1) \neq f(b_k \ldots b_1)$. This would ensure that we never have a spurious collision in both directions simultaneously, also bringing the runtime to $O(N)$ worst-case.