Mean: 114.1; Median: 114.5; Standard deviation: 25.5.
Problem 0. [5 points] Write your name on every page of this exam. (The 20% rule does not apply to this question!)

This exam is calibrated so that you should spend about one minute per point. This question is worth 5 points because we recommend that you spend about 5 minutes skimming all the questions as you write your name.

Problem 1. True or False [32 points] (8 parts)

(a) T F If every edge in a directed acyclic graph has an arbitrary weight, then one can find the simple path from node $s$ to node $t$ of maximum total weight by negating the weight on each edge and running a shortest path algorithm.
Solution: True. First note that the maximum of a sum of variables is the minimum of a sum of their negations. However, for general graphs, this is the longest path problem, which is NP-hard. We can solve this special case only because DAGs are acyclic.

Dijkstra’s does not work since there can be negative edge weights. Bellman-Ford can handle negative edge weights, so run Bellman-Ford. (If there were negative cycles then Bellman-Ford would stop and report “negative cycle” instead of the path, but we know that DAGs can’t have any cycles.)

Some students correctly noted that if you topologically sort the DAG first, you only need one pass of Bellman-Ford; we did not require this level of detail.

(b) T F If every edge in a directed graph has a nonnegative weight, then one can find the simple path from node s to node t of maximum total weight by negating the weight on each edge and running a shortest path algorithm.

Solution: False. This is NP-hard, using a reduction from the longest path problem. Also, note that the original graph may have cycles, which would become negative cycles in the negated graph; no shortest path algorithm can find shortest simple paths in a graph with negative cycles.

(c) T F Assume random variables $X_1, X_2, \ldots, X_n$ are independent and uniform over $[-1, 1]$. By Markov’s inequality, for any $k > 0$,

$$\Pr\left\{ \sum_{i=1}^{n} X_i \geq k \cdot \mathbb{E}\left[ \sum_{i=1}^{n} X_i \right] \right\} \leq \frac{1}{k}.$$ 

Solution: False. In order to apply Markov’s inequality to $\sum_i X_i$, it must be the case that $\sum_i X_i \geq 0$ always. You can also show that, because the variables are uniform over $[-1, 1]$, you have $\mathbb{E}\left[ \sum_{i=1}^{n} X_i \right] = 0$ and $\Pr\left\{ \sum_{i=1}^{n} X_i \geq 0 \right\} = \frac{1}{2}$, which contradicts the statement above.

One common mistake was to say “false because this is not Markov’s inequality.” Although we presented Markov’s inequality differently in class (the expectation was on the right-hand side of the inequality, rather than being inside the probability), the two are equivalent, and the form presented here would work on non-negative variables.

(d) T F If all capacities in a flow network are integers, then the value of the maximum flow must also be an integer.

Solution: True. One way to see this is that the max flow equals the min cut, and the min cut is an integer because it’s the sum of integer edge capacities.
Another way is to argue that, by induction, if the original edge capacities are all integers, then when running the Ford-Fulkerson algorithm, the residual capacities are always integers so every augmenting path augments an integral amount of flow; then by the correctness of the Ford-Fulkerson algorithm, the value of the max flow is an integer.

Many students tried to argue that, in the max flow, the flow along every individual edge is integral. The Ford-Fulkerson algorithm shows that there always exists a solution in which this is true, but it’s possible to find an equivalent optimal solution where the total flow through the graph is integral even though the flow through individual edges are not. Therefore, any argument along these lines is incorrect.

Consider for instance the following graph, where all capacities are 1 and the flow is as shown:

\[ \begin{array}{c}
  & 1/2 & 1/2 \\
 1/2 & & 1/2 \\
 1/2 & & \\
 \end{array} \]

\[ \begin{array}{c}
  s & 1 & t \\
  & & \\
  & & \\
 \end{array} \]

\( (e) \ T \ F \) If there is a polynomial-time reduction from the NP-hard problem A to another NP-hard problem B, and there is also an efficient \( c \)-approximation algorithm to B; then there is also an efficient \( c \)-approximation algorithm for A.

**Solution:** False. In class, it was shown that vertex cover has a 2-competitive approximation algorithm. It was also shown that non-metric TSP has no approximation algorithm at all. However, there is also a poly-time reduction from non-metric TSP to vertex cover. Should mention assumption of \( P \neq NP \).

\( (f) \ T \ F \) The following is correct pseudocode for deleting an element \( x \in \{0, 1, \ldots, u-1\} \) from a van Emde Boas tree \( V \) in \( O(\lg \lg u) \) time:

\begin{verbatim}
DELETE(V, x)
1   DELETE(V.cluster[high(x)], low(x))
2   if V.cluster[high(x)] is empty:
3       DELETE(V.summary, high(x))
\end{verbatim}

**Solution:** False.
It doesn’t update $V_{\text{min}}$, it doesn’t update $V_{\text{max}}$, and upon $V_{\text{min}}$ changing, it doesn’t unstore the new min.

Because of these errors, the algorithm is not correct; furthermore, both recursive calls could take $T(\sqrt{u})$ time, so the recurrence relation is $T(u) = 2T(\sqrt{u}) + O(1)$, which is slower than $O(\lg \lg u)$.

(g) **T**  **F**  If we build a skip list on $n$ elements, then we can search for any element in worst case $O(\lg n)$ time.

**Solution:** False, Skip List runs in expected $O(\lg n)$ time. This is not necessarily always the case since the worst case is $O(n)$.

(h) **T**  **F**  If a graph has a unique shortest path $P$ from node $s$ to node $t$, and has a unique minimum spanning tree $T$, then every edge in $P$ must also be in $T$.

**Solution:** False. Counter-example is a triangular graph with edges of weight 3,4,5.
Problem 2. Short Answer [48 points] (6 parts)

(a) Suppose that you are given a black-box algorithm that can solve linear programs in exactly $n^{3.5}$ milliseconds, where $n$ is the number of variables. Your boss gives you a “mixed-integer” linear program with 5 variables constrained to binary values, and $k$ variables that are only constrained to real numbers. How would you solve this problem efficiently, and about how much time would you expect it to take?

Solution: Just iterate through all 32 possible assignments for the binary-constrained variables, and run the solver on each of those assignments; pick the optimal solution out of those assignments that have feasible solutions. It takes $32 \times k^{3.5}$ milliseconds, plus minute epsilon for picking the optimal solution plus loop overhead. Or if you could run 32 threads at once, this can be trivially parallelized, and takes only $k^{3.5}$ ms plus trivial overhead.
(b) Given a weighted directed graph $G = (V, E, w)$ and two nodes $s, t \in V$, write a linear program whose optimal value is the weight of the minimum $s$-$t$ cut, and justify why this is true.

**Solution:** Look at Max-Flow instead because it is same as min cut by max flow min cut theorem. Each edge is a variable and has a constraint equal to its capacity. And we constrain on skew symmetry and conservation of flow.

\[
\begin{align*}
\text{maximize} & \quad f(s, v) \\
\text{subject to} & \quad f(u, v) = -f(u, v) \\
& \quad \sum f(u, v) = 0 \\
& \quad f(u, v) \leq c(u, v)
\end{align*}
\]

(c) Write the dual of the following linear program:

\[
\begin{align*}
\text{maximize} & \quad 3x - 2y \\
\text{subject to} & \quad x + 5y \leq 4 \\
& \quad x, y \geq 0
\end{align*}
\]

**Solution:**

\[
\begin{align*}
\text{minimize} & \quad 4u \\
\text{subject to} & \quad u \geq 3 \\
& \quad 5u \geq -2 \\
& \quad u \geq 0
\end{align*}
\]

(d) Consider a Markov Chain on the state space $S = \{0, 1, \ldots, T\}$, where $T \in \mathbb{N}^+$. The transition probabilities are given by $\Pr[i \to i + 1] = p < 0.5$ with $p > 0$, and $\Pr[i \to 0] = 1 - p$ for $i \in \{0, 1, \ldots, T - 1\}$, and $\Pr[T \to T] = 1$. Find a stationary distribution of this Markov Chain\footnote{This Markov Chain has been nicknamed the “Tenure Game.” With probability $p$ you take one step towards tenure, or with probability $1 - p$ you lose it all. Once you have tenure, you keep it forever.} and prove that it is stationary.

**Solution:** Let $x \in \{0, 1, \ldots, T\}$ be the initial state of this Markov Chain. Now look at blocks of $T$ time steps. Then the probability that the state at the end of this will be tenure is clearly at least $p^T > 0$. Let us consider $n$ such blocks. Then $\Pr$($\text{final state is tenure}$) $\geq 1 - (p^T)^n$ which goes to 1 as $n \to \infty$. Hence for any initial state we will reach tenure with probability 1. This shows that the state $T$ is a stationary distribution.

\begin{align*}
&\quad \text{(b)} \quad \text{Given a weighted directed graph } G = (V, E, w) \text{ and two nodes } s, t \in V, \text{ write a linear program whose optimal value is the weight of the minimum } s\text{-}t \text{ cut, and justify why this is true.} \\
&\quad \text{**Solution:** Look at Max-Flow instead because it is same as min cut by max flow min cut theorem. Each edge is a variable and has a constraint equal to its capacity. And we constrain on skew symmetry and conservation of flow.} \\
&\quad \text{\[
\begin{align*}
\text{maximize} & \quad f(s, v) \\
\text{subject to} & \quad f(u, v) = -f(u, v) \\
& \quad \sum f(u, v) = 0 \\
& \quad f(u, v) \leq c(u, v)
\end{align*}
\]} \\
&\quad \text{(c) Write the dual of the following linear program:} \\
&\quad \text{\[
\begin{align*}
\text{maximize} & \quad 3x - 2y \\
\text{subject to} & \quad x + 5y \leq 4 \\
& \quad x, y \geq 0
\end{align*}
\]} \\
&\quad \text{**Solution:**} \\
&\quad \text{\[
\begin{align*}
\text{minimize} & \quad 4u \\
\text{subject to} & \quad u \geq 3 \\
& \quad 5u \geq -2 \\
& \quad u \geq 0
\end{align*}
\]} \\
&\quad \text{(d) Consider a Markov Chain on the state space } S = \{0, 1, \ldots, T\}, \text{ where } T \in \mathbb{N}^+. \text{ The transition probabilities are given by } \Pr[i \to i + 1] = p < 0.5 \text{ with } p > 0, \text{ and } \Pr[i \to 0] = 1 - p \text{ for } i \in \{0, 1, \ldots, T - 1\}, \text{ and } \Pr[T \to T] = 1. \text{ Find a stationary distribution of this Markov Chain and prove that it is stationary.} \\
&\quad \text{**Solution:** Let } x \in \{0, 1, \ldots, T\} \text{ be the initial state of this Markov Chain. Now look at blocks of } T \text{ time steps. Then the probability that the state at the end of this will be tenure is clearly at least } p^T > 0. \text{ Let us consider } n \text{ such blocks. Then } \Pr(\text{final state is tenure}) \geq 1 - (p^T)^n \text{ which goes to 1 as } n \to \infty. \text{ Hence for any initial state we will reach tenure with probability 1. This shows that the state } T \text{ is a stationary distribution.} }
\]
(e) Suppose you are given a nonnegatively weighted directed graph \( G = (V, E, w) \), and you have already computed the all-pairs shortest-path weights \( \delta(u, v) \) for all \( u, v \in V \). Now you add a weighted directed edge \((x, y)\) where \( x, y \in V \), to form a graph \( G' = (V, E', w') \) where \( E' = E \cup \{(x, y)\} \). Show how to (re)compute the all-pairs shortest-path weights \( \delta'(u, v) \) in \( G' \) using \( O(V^2) \) time.

**Solution:** The new edge can only influence the length of a shortest path if it is a part of the new shortest path. Thus, for all pairs, we can check whether the known distance of the old shortest path is shorter than the new potential shortest path including the new edge. Updating the shortest path from \( u \) to \( v \) is thus of the form:

\[
\delta(u, v)' = \min(\delta(u, v), \delta(u, x) + w(x, y) + \delta(y, v))
\]

Each of these checks takes constant time and there are \( O(n^2) \) pairs, thus this updates in \( O(n^2) \) time.

(f) We can implement a queue using two stacks \( S_e \) and \( S_d \) as follows. To enqueue \( x \) into the queue, we push \( x \) onto \( S_e \). To dequeue from the queue, we pop and return the top item from \( S_d \). However, if \( S_d \) is empty, we first fill it (and empty \( S_e \)) by popping the top item from \( S_e \), pushing this item onto \( S_d \), and repeating until \( S_e \) is empty. (You do not need to prove that this data structure correctly implements a queue.) Assuming that push and pop operations take \( O(1) \) worst-case time, prove that the enqueue and dequeue operations take \( O(1) \) amortized time (when starting from an empty queue).

**Solution:** Using potential functions, we will show that enqueue and dequeue each take \( O(1) \) amortized time. The potential function used is \( \Phi = 2|S_1| \), where \( |S_1| \) is the number of elements in the first stack. This is always non-negative since there are a non-negative number of elements in the first stack.

For enqueue: \( \Delta\Phi = 2 = O(1) \), and the actual cost is also \( O(1) \).

For dequeue:

\[
\Delta\Phi + \text{actual cost} = \begin{cases} 2|S_1| - 2|S_1| + 1 & : |S_2| = 0 \\ 1 & : |S_2| > 0 \end{cases} = \begin{cases} O(1) & : |S_2| = 0 \\ O(1) & : |S_2| > 0 \end{cases}
\]

We can also use an accounting method by putting two coins on the enqueued item and using those coins to pay for the two pops during the dequeue.
Problem 3. Final Draft [15 points] (2 parts)

Your recent success as the General Manager of the Arizona Ordinals earned you quite a reputation. The team went undefeated and won the Uber Bowl. You’ve now been asked to consult on the drafting process for a basketball team, the Philadelphia 6.046ers. Unfortunately, the goals of the team, as well as the drafting process (which more closely resembles free agency) are different. Specifically, the owner doesn’t care about the skill level of his team. He just wants to build the biggest team possible. In addition, the players are much more fickle. Each player approaches the owner one at a time, announces the positions he’s willing to play, and leaves the meeting signed to a specific position, or unsigned at all, and never returns. It is still the case that every player may be assigned to at most one position, and every position can be filled by at most one player.

Edges of a bipartite graph $G = (L \cup R, E)$ are revealed to you in the following online manner: Each node $u \in L$ is revealed, one at a time, along with a linked list of every edge in $E$ incident to $u$. Upon revealing $u$, you must either:

1. Match $u$ to an unmatched neighbor of $u$ in $R$, and keep this edge forever.

Mean: 10.5; Median: 11.0; Standard deviation: 3.9.
2. Throw away $u$ and never match it.

After processing every node in $L$, you will have some matching, $\text{APX}$, with $|\text{APX}|$ edges. Let $\text{OPT}$ denote the maximum-cardinality matching in $G$.

(a) [5 points] Show that no deterministic algorithm can guarantee $|\text{APX}| > \frac{|\text{OPT}|}{2}$.

**Hint:** Design two graphs $G$ and $G'$ (with the same $L$ and $R$) that are identical after only one node has been revealed, but would require you to choose a different edge. One possible example has $|L| = |R| = 2$.

**Solution:** Let $L = \{a, b\}$ and $R = \{c, d\}$. Put an edge between $(a, c)$ and $(a, d)$ in both $G$ and $G'$. In $G$, put an edge between $b$ and $c$. In $G'$, put an edge between $b$ and $d$. In both graphs, node $a$ is revealed first.

For both $G$ and $G'$, it is clear that the optimal matching has two edges. It is also clear that in $G$, unless we match $a$ to $d$, we can’t get more than one edge. Similarly in $G'$, unless we match $a$ to $c$, we can’t get more than one edge. But because the edges incident to $a$ are identical in both $G$ and $G'$, any deterministic algorithm must make exactly the same choice in both scenarios. If our algorithm chooses to match $a$ to $c$, then we get only one edge in $G$. If our algorithm chooses to match $a$ to $d$, then we only get one edge in $G'$. So no matter what deterministic algorithm we use, for at least one of $\{G, G'\}$ we get only one edge when the optimum is two. So no deterministic algorithm can guarantee $|\text{APX}| > \frac{|\text{OPT}|}{2}$ for all graphs.

(b) [10 points] Design an efficient, deterministic algorithm that guarantees $|\text{APX}| \geq \frac{|\text{OPT}|}{2}$ for all bipartite graphs $G$, prove that it obtains this guarantee, and show that it runs in polynomial time.

**Solution:**

We use the following simple algorithm:

1. Initialize $\text{APX} = \emptyset$.
2. As each node $u \in L$ is processed, check if any neighbors of $u$ are unmatched in $\text{APX}$. If all of $u$’s neighbors are matched in $\text{APX}$, throw $u$ away.
3. Otherwise, pick any unmatched neighbor $v$. Update $\text{APX}$ by matching $u$ to $v$ and continue.

It is clear that this algorithm runs in polynomial time: every time a node is processed, we scan its linked list to see if any neighbors are currently unmatched in $\text{APX}$. Clearly, each edge will only be processed once, and each node will only be processed once, so the total runtime is $O(|V| + |E|)$.

We now show that $|\text{APX}| \geq \frac{|\text{OPT}|}{2}$. We first claim that for every edge $e = (u, v) \in \text{OPT}$, either $u$ or $v$ is matched in $\text{APX}$. If $e \in \text{APX}$ as well, then clearly this is true, if not, then our algorithm chose not to add $e$. Why? When we processed $u$, ...
maybe $v$ was already matched. If so, then clearly $v$ is matched in APX. Or, maybe $v$ wasn’t already matched. But then our algorithm would certainly match $u$ to some node (possibly $v$, possibly not), and $u$ would clearly be matched in APX. Therefore, for every $e = (u, v) \in \text{OPT}$, either $u$ or $v$ is in APX.

Now let’s count the number of nodes in APX: By the argument above, there are at least $|\text{OPT}|$ nodes in APX because every edge in OPT contributes at least one node. Furthermore, no node is double counted because OPT is a matching, and therefore every node can be counted by at most one edge.

So the number of edges in APX is exactly half the number of nodes in APX, which is at least the number of edges in OPT. Therefore, we get $|\text{APX}| \geq |\text{OPT}|$ as desired.

**Common Mistakes:** Most gave a correct algorithm, but few gave a perfect proof. Most proofs looked similar to the proof above, which “charged” all nodes missing in APX from OPT to some node in APX. This part of the argument was usually correct. However, when using this claim to count the number of nodes or edges in APX, one needs to make sure that nothing is double counted. Indeed, this is true because OPT is a matching, but the proof is not complete with this step omitted.
Problem 4. Final Countdown [15 points]

You’re the DJ of 604.6, a cheesy 80s pop radio station. Unfortunately, cheesy 80s pop isn’t as popular as it used to be and you’re in need of some funding to stay afloat. In order to raise funds, you’ve decided to air a countdown of the best cheesy 80s pop one-hit wonders ever. This turns out to be a disastrous idea, as the only people who listen to cheesy 80s pop one-hit wonders are the artists themselves, and they’re all broke. Still, they like to hear their songs on the radio, and if they hear their own song, they’ll donate one dollar. Each artist truly loves listening to cheesy 80s pop, except for a few songs that they just can’t stand. Therefore, they’ll listen to the entire countdown until they hear a song they can’t stand, then they’ll stop. Your manager says you need to raise $k$ dollars or the station will go bankrupt. Your job is to decide whether there exists an ordering of cheesy 80s pop one-hit wonders so that at least $k$ artists will hear their own song.

In the POP problem, you are given an integer $k$, a set $V$ of $n$ artists, and for each artist $v \in V$, you are given the set $E(v)$ of other artists that $v$ hates. Your goal is to output an ordered list $L$ of all $n$ artists. Call an artist $v$ satisfied by $L$ if, in the $L$ ordering, $v$ comes before every artist that $v$
hates (i.e., every element of \( E(v) \)). The POP problem asks whether it is possible to satisfy at least \( k \) artists by some ordered list \( L \).

Prove that POP is NP-complete. You may find the following known NP-complete problem useful, although you are not required to use it:

**INDIE (Independent Set):** Given as input an undirected graph \( G = (V, E) \) and an integer \( k \), call a subset of nodes \( S \subseteq V \) independent if there are no edges between any two nodes in \( S \) (formally, \((u, v) \notin E \) for all \( u, v \in S \)). The Independent Set problem asks whether there exists an independent set of size at least \( k \) in \( G \).

**Solution:**

We first show that POP is in NP. Assume that the answer to a specified POP instance is yes. Then there exists an ordering that satisfies at least \( k \) nodes. So our verifier will take as input an ordering of the \( k \) nodes and check one by one whether each node is satisfied. This clearly takes polynomial time, as there are \( n \) nodes to check, and to see if each node is satisfied requires checking the position of at most \( n \) nodes in the ordering.

Next we prove that POP is NP-hard by designing a Karp reduction from INDIE to POP. Given an input to INDIE—a graph \( G = (V, E) \) and an integer \( k \)—we construct the following POP instance:

1. \( k := k \).
2. \( V := V \).
3. For all \( v \in V \), define \( E(v) \) to be all neighbors of \( v \) in \( G \).

Observe that, in this instance, \( v \in E(u) \) if and only if \( u \in E(v) \). Effectively, we are converting an undirected graph into a directed graph by doubling every edge.

It is clear that this reduction can be implemented in polynomial time. Copying the graph and listing the edges can be done in polynomial time, as well as copying \( k \).

On the one hand, we show that, if the answer to the INDIE instance is yes, then so is the answer to the generated POP instance. Let \( S \) be the independent set in \( G \). Then consider the ordering that puts every element of \( S \) first (in any order), followed by everything else (in any order). Then every element of \( S \) will be satisfied because they don’t hate any node in \( S \), and come before every node not in \( S \). Therefore the elements of \( S \) are a satisfied set of size \( k \).

On the other hand, we show that, if the answer to the generated POP instance is yes, then so is the answer to the given INDIE instance. Consider any ordering, and any satisfied set \( S \) of size \( k \) for that ordering. We claim that \( S \) is an independent set. Assume for contradiction that \( S \) were not an independent set and there were an edge between \( u \) and \( v \) both in \( S \). Then one of \( \{u, v\} \) must come first in the ordering. Without loss of generality, say it is \( u \). Then because there is an edge between \( u \) and \( v \) in \( G \), \( v \) necessarily hates \( u \), and thus is not satisfied, a contradiction. Therefore, \( S \) must be an independent set.
Common mistakes: The biggest common mistake was to do the reduction in the wrong direction (from POP to INDIE), which doesn’t prove NP-hardness of POP. Minor mistakes were to omit key points, e.g., that the reduction is possible in polynomial time, or half of the equivalence argument.
Problem 5. Final Act [15 points]

As part of a traveling magic act, your final trick involves walking across a long stage covered in walls of fire and coming out unharmed! To the audience this looks quite impressive, but the secret is in where the flames lie. You always align the fire walls vertically or horizontally with the stage, and do so in a way so that no two walls touch (so there’s always a gap you can walk through without actually touching the fire, but the audience doesn’t notice this in the dimly lit room). Because you’re part of a traveling act, you have to set up a new stage every time. You’ve noticed that the audience gets more excited the more fire walls there are, so your goal is to get the most fire walls on stage at each venue. Each venue has a preset list of horizontal/vertical segments where they are capable of setting up fire walls.

You are given as input a set $V$ of (possibly intersecting) vertical segments and a set $H$ of (possibly intersecting) horizontal segments. (Two segments intersect if they share any point in the plane.) Design an efficient 2-approximation algorithm for finding the largest non-intersecting subset of $V \cup H$. (A set of segments is non-intersecting if no two segments in the set intersect.) Prove that
your algorithm computes a solution \(\text{APX}\) such that \(|\text{APX}| \geq \frac{1}{2}|\text{OPT}|\), where \(\text{OPT}\) denotes the largest non-intersecting subset of \(V \cup H\).

**Hint:** How would you solve the problem in the special case where \(V = \emptyset\)?

**Solution:** Our approach is to solve very simple special cases of this problem exactly first, and then piece together these solutions into an approximation algorithm. Let’s first start with the case that every line is vertical and shares the same \(x\)-coordinate. It’s easy to see that this is just an instance of interval scheduling: we can ignore the \(x\)-coordinate completely, and two lines intersect if and only if their \(y\)-coordinates intersect. It is clear that the constraints of interval scheduling match exactly the constraints of non-intersecting lines, and that the size of any set of intervals is exactly the number of segments in that set. Therefore, if we run the greedy algorithm (recursively picking the interval with the earliest end time) given in lecture for interval scheduling on this input, we can solve exactly this special case of the problem.

Now, what if every segment is vertical, but not all segments share the same \(x\)-coordinate? For this case, it’s clear that two vertical segments can only intersect if they share an \(x\)-coordinate. So we can partition the input into groups \(S_c\), where \(S_c\) contains all segments with \(x\)-coordinate \(c\). Within each \(S_c\), run the optimal algorithm described in the previous paragraph, and call the output \(L_c\). Output \(\bigcup_c L_c\). It is clear that no non-intersecting set can do better than \(\bigcup_c L_c\), because any intersecting set that beats \(\bigcup_c L_c\) necessarily beats \(L_c\) on some \(S_c\), which is impossible by the correctness of our algorithm in the previous paragraph (which is in turn due to the correctness of the greedy algorithm for interval scheduling). It is also clear that \(\bigcup_c L_c\) is itself non-intersecting. Each \(L_c\) is non-intersecting, and it is impossible for any two vertical segments with different \(x\)-coordinates to intersect. Therefore, the algorithm is correct and exactly optimal.

It is clear that both algorithms above work if we replace “vertical” with “horizontal” as well. So finally, our algorithm for input with both horizontal and vertical segments will be the following: First, let \(H\) denote only the horizontal segments and \(V\) denote only the vertical segments. Run the algorithm of the previous paragraph on both \(H\) and \(V\), and call the output \(L(H)\) and \(L(V)\). Output whichever of \(\{L(H), L(V)\}\) is larger.

To prove that this algorithm obtains a 2-approximation, consider the optimal solution \(L^*\). We can partition \(L^*\) into \(L^*(H)\) and \(L^*(V)\), consisting of the horizontal and vertical segments (respectively) in \(L^*\). Both \(L^*(H)\) and \(L^*(V)\) are non-intersecting solutions that use only the horizontal (respectively, vertical) segments of the input. Therefore, by the correctness of our previous algorithms on parallel input, we must have:

\[
|L^*(H)| \leq |L(H)| \\
|L^*(V)| \leq |L(V)| \\
\Rightarrow |L(H)| + |L(V)| \geq |L^*| \\
\Rightarrow \max\{|L(H)|, |L(V)|\} \geq \frac{1}{2}|L^*|
\]
The last line exactly states that the algorithm is a 2-approximation, as desired. Lastly, it is clear that the algorithm runs in polynomial time. We can split the input into vertical and horizontal segments in poly-time, split the parallel segments up by $x$ (respectively, $y$) coordinate in poly-time. And then we can run the greedy algorithm for interval scheduling in poly-time.
Problem 6. Final Project [15 points]

Ben Bitdiddle is a senior in Course 6, and he wants to make a Photo Album webapp for his 6.UAP. He has already devised a storage method for photos that allows constant-time virtual references for individual photos on the server, and has a neat GUI for users to construct and look through their albums: all that is left is the nasty back-end work, i.e., making his GUI actually functional. Ben is looking for the most efficient way to handle the maintenance of these albums.

Design an efficient data structure that stores a set \( S \) of objects, and maintains a consistent (but arbitrary) ordering \(<\) of the objects. **Consistent** means that, once your data structure decides \( o < o' \) for two objects \( o, o' \in S \), it will never later decide that \( o' < o \). Your data structure must support the following operations:

- **INSERT(o)**: Add object \( o \) to the structure. Assume that \( o \notin S \).
- **DELETE(o)**: Remove object \( o \) from the structure, if \( o \in S \).
- **NEXT(o)**: Move to the next object in the structure after \( o \in S \).

Mean: 8.7; Median: 10.0; Standard deviation: 4.5.
• **PREVIOUS**(*o*): Move to the previous object in the structure before *o* ∈ *S*.
• **SEARCH**(*o*): Determine whether *o* ∈ *S*.

Describe your structure, and give **constant-time** algorithms to support these operations. State which, if any, of your time bounds are amortized and/or expected. (Partial credit will be given for slower solutions.)

**Solution:** There was a clarification to students who asked during the exam about the intention of this question: It was not a requirement that if an element was deleted and re-inserted into the table, it retained its original order. This is because w.r.t. the set *S*, it was a different element, in the sense of the ordering being determined dynamically. Those few students that attempted to solve the alternative (and much harder) problem and failed to meet all cases, but demonstrated clear understanding of how to solve the intended problem, received full credit.

Onto the structure:

The data structure is primarily a re-sizing hash table augmented with a linked list. The hash function is calculated over the reference address. The ordering of the list is strictly the order that the elements have been inserted into the table. Thus,

• **INSERT**(*o*): Add *o* to the hash table; resize if necessary. Add *o* to the end of the linked list.
• **DELETE**(*o*): Lookup and delete *o*’s entry from the hash table, but follow its reference pointer and update linked list references.
• **NEXT**(*o*): Follow *o*’s reference pointer and return the next object in the linked list.
• **PREVIOUS**(*o*): Follow *o*’s reference pointer and return the previous object in the linked list.
• **SEARCH**(*o*): Lookup whether *o* is in the hash table.

**NEXT** and **PREVIOUS** can be considered deterministic constant time based on building the nodes around the reference objects, but expected constant time was also accepted for full credit, for two reasons: it was not clear in the problem that you can work under an computational model in which this type of object-building is possible, and we failed to note that you may assume that *o* is already present in the table. Thus, **SEARCH** is the primary cause of requirement of the hash table; some students tried to additionally augment each of the objects’ nodes with a boolean value that is set to true when inserted and false when deleted; however, without the table, this is insufficient, because if an object has never been inserted into the table before, its fields would be garbage data that could appear “true”.

**INSERT**, **DELETE**, and **SEARCH** all **must be expected** constant time, but at least **INSERT** must also be amortized expected constant time, because table doubling is required to maintain all elements with low collision probability. A solution of this structure that noted all runtime features received full credit, but no points were deducted for including **DELETE** as an amortized constant time operation. While we did not ask for this, it does save space, possibly a **lot** of space, which would be important if you wanted this to be a real-world application.
Problem 7. Final Destination(s) [20 points] (3 parts)

You are trying to decide whether two horror movies have identical story lines. Fortunately, each story line has a simple tree-like structure: there is a distinct starting point and the plot diverges along separate paths at various later points. Two story lines are deemed identical if their plots diverge in the same ways after their respective starting points.

For two rooted, unordered trees $T_1$ and $T_2$ each on $n$ nodes, call $T_1$ and $T_2$ interchangeable if there exists a one-to-one correspondence $f$ from the nodes of $T_1$ to those of $T_2$ such that

1. the root $r_1$ of $T_1$ is mapped to the root $r_2$ of $T_2$;
2. $v$ is a child of $w$ in $T_1$ if and only if $f(v)$ is a child of $f(w)$ in $T_2$.

Associate a polynomial $P_v$ with each vertex $v$ recursively as follows. At the base case, a leaf vertex $v$ has associated polynomial $P_v = x_0$. For an internal vertex $v$ of height $h$ having children $v_1, v_2, \ldots, v_k$, its associated polynomial is

$$P_v = (x_h - P_{v_1})(x_h - P_{v_2}) \cdots (x_h - P_{v_k}).$$
Note that there is exactly one variable for each level in the tree.

(a) [8 points] Prove that the degree of $P_{r_i}$ is at most $n$. (The degree of a polynomial is the maximum degree of its terms. The degree of a term $x_1^{\alpha_1}x_2^{\alpha_2} \cdots x_n^{\alpha_n}$ is $\alpha_1 + \alpha_2 + \cdots + \alpha_n$. For example, the degree of $x_1^2x_2 + x_2x_3^3$ is $\max\{2 + 1, 1 + 3\} = 4$.)

Solution: Let $\deg(P_v)$ denote the degree of $P_v$. We claim that $\deg(P_v)$ is at most the size (in terms of the number of nodes) of the sub-tree rooted at $v$. Therefore, the degree of $P_{r_i}$ is at most $n$, the total number of nodes in $T_i$.

We prove this claim by (strong) induction on the height of the node, $h$. The base case is when $h = 0$. In this case, we just have a single leaf node. The polynomial associated with the node is $x_0$, which has a degree of 1. So the base case is true. Now assume that the claim holds for all $h \leq k$, where $k \geq 0$. Let the node $v$ have height $h = k + 1$.

We show that the claim holds for $P_{r_i}$ as well. By the definition of $P_v$ given above, $\deg(\prod_{i=1}^k P_{v_i})$ must be no less than $\deg(P_v)$, because $x_h$ has degree 1 whereas each $P_{v_i}$ has degree at least 1. We can write this degree as follows,

$$\deg(\prod_{i=1}^k P_{v_i}) = \sum_{i=1}^k \deg(P_{v_i}).$$

Since each child node $v_i$ has height no greater than $k$, by the inductive hypothesis, $\deg(P_{v_i})$ is at most the size of the sub-tree rooted at $v_i$. Therefore, $\deg(P_v)$ is at most the sum of the sizes of the sub-trees rooted at its child nodes, this is in term bounded by the size of the tree rooted at $v$.

Alternate approach: in fact, a tighter bound can be shown: $\deg(P_v)$ is exactly the number of leaf nodes in the sub-tree rooted at $v$. Since the total number of leaf nodes in $T_i$ is at most $n - 1$, $\deg(P_v)$ is at most $n$. This tighter bound can be proved similarly by induction or by rigorously showing that, after expanding $P_v$, the term whose variables are all $x_0$ has the same degree as $P_v$.

Common mistakes: Most students had the right approach: trying to bound $\deg(P_v)$ by the size of the sub-tree or the number of leaves in the sub-tree. This is worth three points. The other five points are awarded based on the completeness of the proof. Some students provided the right observation that a tree with a large branching factor closer to the root would have a larger degree; but they didn’t give a formal proof. In this case, a score between three and five is given based on how precise the observation is stated/demonstrated.

(b) [8 points] Prove that $P_{r_1} = P_{r_2}$ (for all values of $x_1, x_2, \ldots$) if and only if $T_1$ and $T_2$ are interchangeable.

Solution: We prove this claim from both directions. First, assume that $P_{r_1} \equiv P_{r_2}$. We know that the two polynomials must have the same variable $x_H$ in the last recursive
step of their constructions. Since $H$ corresponds to the height of the vertex at the recursive step, and the root of the tree always has the greatest height, this identified $r_1$ with $r_2$. Now we have

$$P_{r_1} = \prod_{v \in \delta(r_1)} (x_H - P_v) \quad \text{and} \quad P_{r_2} = \prod_{u \in \delta(r_2)} (x_H - P_u).$$ (1)

Each $x_H - P_v$ and $x_H - P_u$ are irreducible factors in the unique factorization domain $\mathbb{F}[x_H]$ (this is analogous to the unique-prime-factorization theorem). So $P_{r_1} \equiv P_{r_2}$ implies that there is a one-to-one mapping $f$ from $\delta(r_1)$ to $\delta(r_2)$ such that $x_H - P_u \equiv x_H - P_v$, or $P_u \equiv P_v$, if $f(u) = v$. Now we can apply the same argument above (for $P_{r_1}$ and $P_{r_2}$) to these pairs of equivalent sub-multinomials. The union of the mappings we define at each step will be an isomorphism $f$.

On the other hand, if $T_1$ and $T_2$ are isomorphic, then the heights of the roots of both trees must be the same; denote it by $H$. So again we have the recursions from (1) above. Furthermore, the recursive constructions of $P_{r_1}$ and $P_{r_2}$ at each step must be the same. So once we chase out the two recursions, we will see that $P_{r_1} \equiv P_{r_2}$.

**Common mistakes:** Some students only proved the claim in one direction; each direction is worth four points. In the “polynomial identity to tree isomorphism” direction, many students neglected to mention how to identify the roots of the two trees given just the two polynomials. This step is required for constructing a valid underlying isomorphism $f$.

(c) [4 points] Give a linear-time algorithm for testing whether two given rooted trees are interchangeable. Your algorithm should be correct with probability at least $\frac{2}{3}$.

**Hint:** Assume (a) and (b), and use the following theorem of Schwartz and Zippel.

**Theorem:** Let $P$ be a non-zero multivariate polynomial in $x_1, x_2, \ldots, x_n$ of degree $d$. If $r_1, r_2, \ldots, r_n$ are chosen uniformly at random from $\{1, 2, \ldots, m\}$, then

$$\Pr \left\{ P(r_1, r_2, \ldots, r_n) = 0 \right\} \leq \frac{d}{m}.$$  

**Solution:** So now we devise a randomized algorithm to test if $P_{r_1} \equiv P_{r_2}$. By Schwartz-Zippel, we uniformly choose $x_i, i \in \{0, 1, \ldots, H\}$, from $\{1, 2, \ldots, m\}$ and evaluate $P_{r_1} - P_{r_2}$ on this random point to see if it’s 0. If the evaluation is not zero, by part (b), $T_1$ and $T_2$ cannot be isomorphic, so we output False; if the evaluation is zero, we output True. We analyze the algorithm below.

As in part (a), let $c(v)$ be the number of child nodes for each vertex $v$. The above evaluation takes $c(v)$ unit-cost steps at each $P_v$. So the total cost of this evaluation is the number of edges in the tree, which is linear in $n$. As shown in part (a), the degree of $P_{r_i}$ is at most $n$. Therefore, as long as we choose $m$ to be strictly larger than $n$, the false-positive probability will be $\frac{n}{m} < 1$. We can improve the probabilistic guarantee
of our algorithm by either increasing the size of \( m \) (by some constant multiplicative factor) or by repeating the algorithm a constant number of times. For instance, setting \( m = 3n \) works.

**Common mistakes:** Many students did not show that evaluating \( P_{r_1} - P_{r_2} \) on the random point can be done in linear time. This is worth one point.
Problem 8. Final Boss [15 points]

Mean: 6.1; Median: 3.0; Standard deviation: 4.5.

After many hours of playing Xelda: Brilliant Blade, the latest installment in the series, you have finally made it to the final boss battle with the evil wizard Cannon. His signature move involves climbing onto a chandelier attached to the ceiling and diving at you. Since the chandelier is very high above the ground, this attack can be very dangerous if it connects. Your first priority, therefore, is to ensure that he cannot perform this move.

You notice that the chandelier is attached to the ceiling by a complex network of enchanted chains. The chains are impenetrable, but you notice that whoever put the chains together doesn’t know how to weld properly. Wherever two or more chains meet up, or wherever a chain is connected to the chandelier or the ceiling, they are connected by a weak, brittle welding joint. You figure that if you slash your sword through one of these joints, the joint will shatter, effectively disconnecting all chains that were attached to it.

However, not all joints have the same brittleness. Your companion, Phi, can use her magic powers to determine how easy it would be to break through each joint. Armed with this knowledge, you
set out to dislodge the chandelier from the ceiling while minimizing the amount of work you need to do.

You are given a directed graph \( G = (V, E) \), two disjoint subsets \( A, B \subseteq V \), and a vertex weight function \( w : V \rightarrow \mathbb{R}^+ \). Call a subset of vertices \( S \subseteq V \) winning if the removal of \( S \) disconnects all vertices in \( A \) from all vertices in \( B \), i.e., every path in \( G \) from a vertex in \( A \) to a vertex in \( B \) must contain at least one vertex in \( S \). Give a polynomial-time algorithm to find a winning set of vertices of minimum total weight.

(Note that \( A \cup B \) is not necessarily \( V \), and that \( S \) may contain vertices that are in \( A \) or \( B \).)

**Solution:** We will construct a flow network \( G_f = (V_f, E_f) \) as follows:

- Create a source \( s \) and a sink \( t \) in \( V_f \).
- For each vertex \( v \in V \), create two vertices \( v_{in}, v_{out} \) in \( V_f \), and create an edge \((v_{in}, v_{out})\) in \( E_f \) with capacity \( w(v) \).
- For each edge \((u, v)\in E\), create an edge \((u_{out}, v_{in})\) in \( E_f \) with infinite capacity.
- For each edge \( a \in A \), create an edge \((s, a_{in})\) in \( E_f \) with infinite capacity.
- For each edge \( b \in B \), create an edge \((b_{out}, t)\) in \( E_f \) with infinite capacity.

Any finite cut through this network may only pass through \((v_{in}, v_{out})\) edges, because they are the only edges of finite capacity. Furthermore, we can create a one-to-one correspondence between finite minimum cuts and winnings sets of vertices \( S \) by placing \( v_{in} \) and \( v_{out} \) in different sets for each \( v \in S \). We know that there exists a finite \( s-t \) cut, because we can just cut all edges \((a_{in}, a_{out})\) for each \( a \in A \), so the minimum cut of the network will also be finite. Therefore, it suffices to find a minimum cut of \( G_f \), and that will give us our answer.

We discussed an algorithm in lecture to compute a minimum cut of a network while proving the max-flow-min-cut theorem. The algorithm is as follows: Let \( f \) be a maximum flow of the network, and let \( S_f \) be the set of all vertices reachable from \( s \) in the network’s residual graph. Then \( S_f, V_f - S_f \) is a cut of minimum weight of \( G_f \).

To solve our original problem, find a maximum flow \( f \), and use DFS to compute \( S_f \) as defined in the previous paragraph. We know that \( f \) must be finite, since any path from \( s \) to \( t \) in \( G_f \) must pass through \((a_{in}, a_{out})\) for some \( a \in A \), and the total weights of all such edges is finite. Therefore no edge of infinite capacity can be saturated, so we are guaranteed to stop expanding \( S_f \) at an edge with finite weight. The only such edges are those that replaced the vertices, so for several \( v \in V \) we have \( v_{in} \in S_f \) and \( v_{out} \notin S_f \). Since \( S_f, V_f - S_f \) is a minimum cut, \( S = \{v \in V : v_{in} \in S_f, v_{out} \notin S_f \} \) is a winning subset of vertices with minimum total weight.

The conversion can be done in \( O(V + E) \) time, and finding a maximum flow can be done in \( O(V_f E_f^2) = O(V(V + E)^2) \) time using Edmonds-Karp. Since the graph is connected, we know that \( E \) cannot be asymptotically smaller than \( V \), so we may safely rewrite this as \( O(VE^2) \). Doing the DFS to find \( S_f \) and computing \( S \) takes \( O(V + E) \) time, so our final running time is \( O(VE^2) \).
If you wanted, you could do better with Push-Relabel or Relabel-to-Front and get $O(V^2 E)$ or $O(V^3)$, respectively. (Some students simply stated that finding a maximum flow could be done in polynomial time instead of doing a full running-time analysis. Since we only asked for “a polynomial-time algorithm,” we did not deduct points for this, as long as the student explicitly said that the algorithm could run in polynomial time.)

**Common mistakes:** Many students forgot to elaborate on how to make the connection between minimum cuts of the network and answers to the original problem. Several students also tried solving this using dynamic programming without much progress or removing nodes greedily in a certain order.