CPSC 320 2025S: Assignment 2

This assignment is due **Friday, July 18 at 7 PM**. Late submissions will not be accepted. Please follow these guidelines:

- Prepare your solution using LaTeX and submit a pdf file. Easiest will be to submit using the .tex file provided. For questions where you need to select a circle, you can simply change `\fillinMCmath` to `\fillinMCmathsoln` .
- Enclose each paragraph of your solution with `\begin{soln}Your solution here...\end{soln}`. Your solution will then appear in dark blue, making it a lot easier for TAs to find what you wrote.
- Submit the assignment via GradeScope. Your group must make a **single** submission via one group member's account, marking all other group members in that submission **using GradeScope's interface**.
- After uploading to Gradescope, link each question with the page of your pdf containing your solution.

Before we begin, a few notes on pseudocode throughout CPSC 320: Your pseudocode should communicate your algorithm clearly, concisely, correctly, and without irrelevant detail. Reasonable use of plain English is fine in such pseudocode. You should envision your audience as a capable CPSC 320 student unfamiliar with the problem you are solving. You may **neither** include what we consider to be irrelevant coding details **nor** assume that we understand the particular language you chose. (So, for example, do not write `#include <iostream>` at the start of your pseudocode, and avoid language-specific notation like C/C++/Java's ternary (question-mark-colon) operator.)

Remember also to **justify/explain your answers**. We understand that gauging how much justification to provide can be tricky. Inevitably, judgment is applied by both student and grader as to how much is enough, or too much, and it's frustrating for all concerned when judgments don't align. Justifications/explanations need not be long or formal, but should be clear and specific (referring back to lines of pseudocode, for example). Proofs should be a bit more formal.

On the plus side, if you choose an incorrect answer when selecting an option but your reasoning shows partial understanding, you might get more marks than if no justification is provided. And the effort you expend in writing down the justification will hopefully help you gain deeper understanding and may well help you converge to the right selection :).

Ok, time to get started...

Group Members

Please list the CWLs of all group members here (even if you are submitting by yourself). We will deduct a mark if this is incorrect or missing.

1 Take Your Order

[10 points] Take the following functions and arrange them in ascending order of growth rate. That is, if $g(n)$ appears after $f(n)$ in your ordering, it should be the case that $f(n) \in O(g(n))$. (Example: the functions n and n^2 would appear in the order: n, n^2.) Justify your answer.

$$f_1(n) = n$$
$$f_2(n) = 2^n$$
$$f_3(n) = \sqrt{n}\log n$$
$$f_4(n) = \frac{n \log n}{\log(\log n)}$$
$$f_5(n) = (n-2)!$$
$$f_6(n) = n^{\lg n}$$

We first develop the following lemma: if $f \in O(g), g \in O(h)$ then $f \in O(h)$.

Suppose $f \in O(g), g \in O(h)$. Then there exists $c_1, c_2 \in \mathbb{R}^+, n_1, n_2 \in \mathbb{N}$ so that

$f(n) \leq c_1 \cdot g(n)$ whenever $n \geq n_1$ and $g(n) \leq c_2 \cdot h(n)$ whenever $n \geq n_2$.

Then we have that $f(n) \leq c_1 c_2 \cdot h(n)$ whenever $n \geq \max(n_1, n_2)$. So, $f \in O(h)$.

Claim: The orderding is $(f_3, f_1, f_4, f_6, f_2, f_5)$.

It suffices to show for each neighbour pair (f_i, f_j) in the order that $f_i \in O(f_j)$ to get if g appears after f then $f \in O(g)$ by the lemma.

1. $f_3 \in O(f_1)$. We show that $\lim_{n\to\infty} \frac{f_1(n)}{f_3(n)}$ diverges.

   So, $\frac{f_1(n)}{f_3(n)} = \frac{n}{\sqrt{n}\log(n)} = \frac{\sqrt{n}}{\log(n)}$.

   Without loss of generaltity, assume the base is e, then using real valued limits and L'Hopital's rules we get:

   $$\lim_{x\to\infty} \frac{\sqrt{x}}{log(x)} =^H \lim_{x\to\infty} \frac{1}{2} \cdot \frac{1}{\sqrt{x}} \cdot \frac{1}{\left(\frac{1}{x}\right)} = \lim_{x\to\infty} \frac{\sqrt{x}}{2} = \infty.$$

   And thus, we get that $f_3 \in O(f_1)$.

2. $f_1 \in O(f_4)$. We make use of the fact that $\log(n) \in O(n)$ and that $\log(n)$ is strictly increasing.

   So then for some $c \in \mathbb{R}^+$ and $n_1 \in \mathbb{N}$, we have that $\log(n) \leq cn$ whenever $n \geq n_1$.

   Applying the log function to both sides, we get $\log(\log(n)) \leq \log(c) + \log(n)$.

   Then we see that $\log(c) + \log(n) \leq 2\log(n)$ whenever $n \geq \lceil c \rceil$.

   Thus, we have that $\log\log(n) \leq 2\log(n)$ whenever $n \geq n_2$, where $n_2 = \max(n_1, \lceil c \rceil)$.

   We know that $n \geq 1$, and hence we multiply both sides to get $n\log\log(n) \leq 2n\log(n)$.

   Rearranging we get, $n \leq 2 \cdot \frac{n\log(n)}{\log\log(n)}$ whenever $n \geq n_2$, thus $f_1 \in O(f_4)$.

3. $f_4 \in O(f_6)$. We assume that the base is $b > 1$.

   First, observe $\log(n) > b$ whenever $n \geq \lceil b^b \rceil + 1$. And so, $\log\log(n) > 1$ and thus $\frac{1}{\log\log(n)} < 1$.

   Then we get that $\frac{n\log(n)}{\log\log(n)} \leq n\log(n)$. And so, it suffices to show $n\log(n) \in O(n^{\lg(n)})$.

   Since the base does not matter, is also suffices to show that $\lg(n) \in O(n^{\lg(n)-1})$.

   We will use a limit argument to show that it is the case. So then using real valued limits and L'Hopital's we get

$$\lim_{x\to\infty} \frac{x^{\lg(x)-1}}{\lg(x)} =^H \frac{1}{\ln(2)} \lim_{x\to\infty} x^{(\lg(x)-2)}(2\ln(x)-\ln(2)) \frac{1}{\left(\frac{1}{x\ln(2)}\right)} = \lim_{x\to\infty} x^{\lg(x)-1}(2\ln(x)-\ln(2)) = \infty.$$

   Then $\lg(n) \in O(n^{\lg(n)-1})$. More formally, we can then write:

$$\exists c_1 > 0, \exists n_1 \in \mathbb{N}, \forall n \geq n_1, \lg(n) \leq c_1 \cdot n^{\lg(n)-1}.$$

   Then if we set $c_2 = c_1 \cdot \lg(b)$, we have

$$\forall n \geq n_1, \log(n) \leq c_2 \cdot n^{\lg(n)-1} \implies n\log(n) \leq c_2 \cdot n^{\lg(n)}.$$

   Now, if $n_2 = \max(\lceil b^b \rceil + 1, n_1)$, we can write

$$\forall n \geq n_2, \frac{n\log(n)}{\log(\log(n))} \leq c_2 \cdot n^{\lg(n)}.$$

   Hence, $f_4 \in O(f_6)$.

4. $f_6 \in O(f_2)$. $f_6 = n^{\lg(2)} = 2^{\lg(n^{\lg(2)})} = 2^{\lg^2(n)}$. $f_2 = 2^n$.

   It suffices to show $\lg^2(n)$ is bounded by n since the exponential with base 2 is an increasing function.

   We will again use a limit argument, so again we tak real valued limits,

$$\lim_{x\to\infty} \frac{x}{\lg^2(x)} =^H \frac{2}{\ln(2)} \lim_{x\to\infty} \frac{x}{\lg(x)} =^H \frac{2}{\ln^2(2)} \lim_{x\to\infty} x = \infty.$$

   Thus, $f_6 \in O(f_2)$.

5. $f_2 \in O(f_6)$. To show 2^n is bounded by $(n-2)!$, we show $2^n \leq (n-2)!$ for $n \geq 8$ using induction.

   Base case: $n = 8$: $2^n = 2^8 = 256 \leq 720 = 6! = (8-2)! = (n-2)!$. Assume true for $k \in \mathbb{N}$ with $k \geq 8$.

   Notice, $k - 1 \geq 7 > 2$. Now, $2^{k+1} = 2 \cdot 2^k \leq 2 \cdot (k-2)! < 7 \cdot (k-2)! \leq (k-1)(k-2)! = (k-1)!$.

   Hence, induction makes it true for $n \geq 8$. Thus, $2^n \in O((n-2)!)$.

2 All-Stars

Suppose you're organizing a tennis tournament between n players which we simply label as $1, 2, \ldots, n$. Each player competes against every other player, and there are no repeat matches and no tied scores. We model the results of the competition as a directed graph with n vertices and exactly one directed edge between each pair of vertices: $G = (V, E)$, with $V = \{1, 2, \ldots, n\}$ where either $(i, j)) \in E$ (if i defeated j), or $(j, i)) \in E$ (if j defeated i). Such a graph is called a *tournament*.

You want to determine a way to rank these players, but a challenge here is that cycles **may** exist in your tournament – e.g., i may defeat j in a match, and j could defeat k, but then we could still have k win a match against i; this would lead to a cycle of length three.

A ranking for the tournament is an ordering $(r_1, r_2, \ldots, r_n)$ where node r_1 is established as the overall rank 1 player. Due to the potential existence of cycles it is challenging to find a fair ranking.

1. [4 points] We decide that the top ranked player should always be some node whose out-degree in the directed graph is maximum. We motivate this claim by proving that if player i has maximum degree, then for every other player j either i beat j, or i beat some player k which beat j. Prove this claim.

   Hint: A proof by induction on n works well.

   Let $G = (V, E)$ be a graph such that it is a tournament. Let $i \in V$ be the node such that it has maximum out-degree.

   We aim to prove that $\forall j \neq i \in V$ either i beat j or i beat $k \in V$ who beat j for any size $|V| = n \in \mathbb{N}$.

   Base case: $|V| = 2$. Then if i has max out-degree then $E = \{(i, j)\}$. Thus, the claim holds.

   Assume the claim holds for $|V| = n$, with $n \in \mathbb{N}$.

   Consider a tournament with $|V| = n + 1$ nodes. Denote the player who has maximum out-degree by i.

   Then remove $j \neq i$ from this graph, so that we have n nodes, denote $V' = V \setminus \{j\}$, with i beat j.

   Either i has the maximum out-degree in this graph or not. We consider each case.

   - Case i has maximum out-degree in V'. Then, by assumption for each node other node $j' \in V'$, either i beat j' or i beat k' who beat j'. Since $|V'| = n$.
     By adding back in j, then the statement holds for $|V| = n + 1$, since i beat j by assumption.
   - Case i does not have maximum out-degree in V'.
     So, by removing j, every other node decreased their out-degree by at most one.
     By assumption i beat j, so its out-degree must have decreased by one in V'.
     Thus, there was some node $k \in V$ such that k had the same out-degree as i, but k did beat j.
     Thus, j beat k. By assumption for each other $j' \in V'$, either k beat j' or k beat k' who beat j'.
     We now add back j to the graph. It remains that we need a connection to j, as k did not beat j.
     Indeed if k beats i who beat j, then assumption holds for $|V| = n + 1$, using k in the hypothesis.
     However, if k did not beat i then we consider removing i from this graph.
     Since k lost to i, then the out-degree of k cannot decrease. So k must have the max out-degree.
     This graph being size n, means there is some $v \in V \setminus \{i\}$ so that k beats v and v beats j.
     Returning i to the graph, the statement holds for $|V| = n + 1$, again using k in the hypothesis.

   By induction, the statement holds for any tournament with $|V| = n \in \mathbb{N}$.

2. [2 points] We decide that a reasonable definition is suggested by the previous result. A sequence of nodes $(r_1, r_2, \ldots, r_n)$ is a *valid ranking* if for each $i = 1, 2, \ldots, n$ the node r_i has maximum out-degree in the sub-tournament induced by the nodes $r_i, r_{i+1}, \ldots, r_n$. Give a polynomial-time algorithm which decides if a given permutation of the nodes $(v_1, v_2, \ldots, v_n)$ is a valid ranking.

   Let $P = (v_1, v_2, \ldots, v_n)$ be a permutation on V.

   We will assume that we have an adjency matrix to represent the edges in the graph M.

```
1:  procedure VALID-RANKING(P, M)
2:      for each i = 1, 2, ..., n − 1 do
3:          store the sum from j = i to j = n of M(j) as d_i
4:          for each j = i + 1, ..., n − 1 do
5:              store the sum from k = j to k = n of M(k) as d_k
6:              if d_i < d_k then
7:                  end the procedure, report it is not valid
8:              end if
9:          end for
10:     end for
```

11: report the permutation as valid
12: **end procedure**

This algorithm is $O(n^3)$, where $n = |P|$. In the worse case, we have a valid permutation, and the algorithm does work as follows.

We have to iterate through the permutation list $n-1$ times. Access to the elements will be $O(1)$ as we assume we are given an array.

Then for each iteration, we are iterating through the entire adjency matrix, except each iteration we are considering one less node.

Access to the adjency matrix is $O(1)$ thus the iteration of the adjency matrix is $O(i^2)$.

Then, for each i, the work we are doing is i^2. Thus, summing $i = 1, 2, \ldots, n-1$ gives us $O(n^3)$.

3. [2 points] Describe a tournament of size n which has precisely one valid ranking.

   Suppose we are given a tournament graph $T = (V, E)$ with a vertex ordering $v_1, v_2, \ldots, v_n$ such that for all $i < j$, the directed edge (v_i, v_j) is in E. That is, every vertex points to all vertices that come after it in the order.

   Suppose we have a tournament graph $T = (V, E)$ such that there is an ordering, $v_1, v_2, \ldots, v_n$ so that there is an edge (v_i, v_j) iff $i < j$.

   This would create one valid ranking, and this ordering is exactly that ranking.

   Before we prove that statement, we will show it is a tournament in the first place, and that is is valid ranking.

   First, we require that every player has played someone else and only once or in other words, either $(v_i, v_j) \in E$ or $(v_j, v_i) \in E$ but not both.

   Let $v_i, v_j \in V$ such that $i \neq j$. Then either $i < j$ or $j < i$. Then we can only have one of the edge pairs in E by assumption.

   Next, we require that v_i has maximum out-degree in its subtournaments.

   Observe that for any v_i in a subtournament it's out degree is $d_i = n - i + 1$, independent of the sub tournament.

   Then for any $i < j$, we see that $n - j + 1 < n - i + 1$, but this precisely means that $d_j < d_i$, thus it is valid ranking.

   Proof: For contradiction, suppose that there were two valid rankings.

   Then it must be some permutation on $P := v_1, v_2, \ldots, v_n$.

   Then there exists some $i < j$ in P such that $j < i$ in P'. But this means that $d_j < d_i$ in the subtournament for j in P'.

   This contradicts that P' was a valid ranking.

4. [4 points] One brute force method for producing all valid rankings is to consider all permutations of the nodes and then use your algorithm for Part b to check each one if it is a valid ranking. This will run in time $O(n!p(n))$ where $p(n)$ is the runtime of your algorithm in Part b.

   Instead, create a better algorithm which runs in time $O(|Val|poly(n))$ where Val is the set of valid rankings, and $poly(n)$ is a polynomial in n. (You do not need to analyze the runtime of your algorithm.)

   Find the set S_0 such that each node in S_0 has equal out-degree and is the greatest amongst all V.

   Find the set S_1 such that each node in S_1 has equal out-degree and is the greatest amongst all $V \setminus S_0$.

   Find set S_k so that each node in S_k has equal out-degree and is the greatest amongst all $V \setminus \bigcup_{i=0}^{k-1} S_i$.

Continue until you have a collection, such that $\bigcup_{i=0}^{k} S_i = V$.

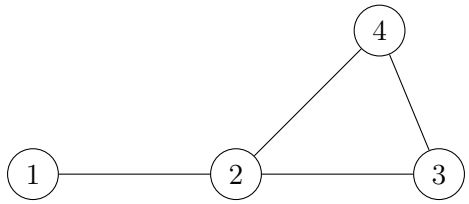
Then create a set of permutation, $P_0, P_1, \ldots, P_k$ such that P_i contains all possible permuations of ordered tuplets on vertices in S_i.

Next, create another set containing all ordered tuples of $r = (p_1, p_2, \ldots, p_k)$ where each $p_i \in P_i$.

Return $R = \{\text{all possible } r\}$ as the set all possible valid rankings.

3 Colour Me Confused

Given a graph $G = (V, E)$, we consider the problem of assigning a colour to each vertex in G such that no two adjacent vertices share a colour. Recall that the *degree* of a vertex $v \in V$ is the number of edges incident on v. Let d be the *maximum degree* in the graph.



The maximum degree d in the graph above is 3. The graph has a 3-colouring (by using different colours for nodes 1, 2, and 3 and colouring node 4 the same as 1 or 2) and a 4-colouring (by using a different colour for every node), but does not have a 2-colouring.

1. [2 points] For any value of d, describe a graph with maximum degree d that can be coloured with two colours.

   Let V be a vertex set. Add $v \in V$. Then also add $v_1, v_2, \ldots, v_n$ to V.

   Then for $v_1, v_2, \ldots, v_d$ add $(v, v_i) \in E$ for $i = 1, 2, \ldots, d$.

   Then colour the node v blue, and colour all other nodes in V red.

   By construction, the degree of all other nodes that are not v is 1 since it's only connected to v.

   Thus, the degree of v is the maxmimum degree of the graph, which is d.

   Since no nodes are adjacent except for ones to v. Then the colour red is never shared by adjacent vertices.

   Thus, this graph with maximum degree d can be coloured with two colours.

2. [2 points] Consider the following greedy algorithm to find a colouring for a graph:

   > Order vertices arbitrarily. Colour the first vertex with colour 1. Then choose the next vertex v and colour it with the lowest-numbered colour that has not been used on any previously-coloured vertices adjacent to v. If all previously-used colours appear on a vertex adjacent to v, introduce a new colour and number it, and assign the new colour to vertex v.

   Prove that this algorithm uses at most $d + 1$ colours.

   Proof: We provide a proof by induction on the number of nodes, $n \in \mathbb{N}$, for graph $G = (V, E)$.

   Base case $|V| = 1$. The highest degree can be $d = 0$, since the edge set would be empty.

   Thus, the algorithm uses $1 = d + 1$ colours to colour the single node and the base case holds.

   Assume for any graph with n nodes that the algorithm will colour any ordering of nodes using at most $d + 1$ colours.

Consider a graph $G = (V, E)$ with $|V| = n + 1$ nodes with the highest degree of any node is d.

Let $v_1, v_2, \ldots, v_n, v_{n+1}$ be any ordering of the nodes. Remove v_{n+1} from this order, and the graph.

Denote the deleted graph without v_{n+1} by G'. Notice the degree of any node in G' can only decrease.

Thus, the maximum degree of G' remains to be d with the removal of v_{n+1}.

By assumption, we can colour the ordering $v_1, v_2, \ldots, v_n$ using at most $d + 1$ colours.

Since v_{n+1} has degree at most d then it has at most d neighbours. Then we add back v_{n+1}.

The number of colours used to colour its neigbours can be at most d, if each are coloured distinctly.

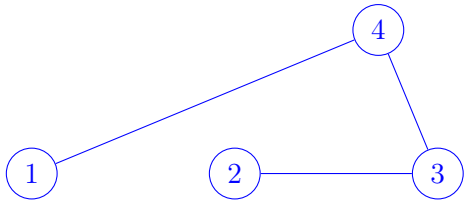
Thus, there remains at least 1 unique colour to colour v_{n+1} so that it is a valid colouring.

In other words, the algorithm uses at most $d + 1$ colours to colour the ordering $v_1, v_2, \ldots, v_{n+1}$.

Induction makes the claim holds true for any graph with n nodes and maximum degree d.

3. [2 points] Give and briefly explain an example where this algorithm does not produce an optimal colouring (i.e., where it uses more colours than is necessary to colour the graph).

   We consider the graph $G = (V, E)$ with $V = \{1, 2, 3, 4\}$ and $E = \{(1, 4), (4, 3), (3, 2)\}$.



   This graph can be coloured with three colours. Namely we can assign $A = \{1, 3\}$ $B = \{2, 4\}$.

   We see that no edges are shared between any vertices in A, B so this is a valid colouring.

   Now consider the ordering $1, 2, 3, 4$. We first colour 1 blue. And then we consider 2.

   There is no edge between $1, 2$ so we colour 2 blue and consider 3.

   So, its adjacent vertices have been coloured with blue, then we introduce a new colour for it red.

   Now we finally consider 4, its neighbours have been coloured with both red and green, so we must introduce a new colour for it purple.

   We have thus coloured this graph using three colours through the greedy algorithm when we could have used two.

4. [5 points] Now, assume that there is at least one vertex v in V with degree **less than** d. Design a greedy algorithm that will colour vertices of G with at most d colours. You should proceed by first **ordering the vertices** in some way, and then assigning colours using the colouring strategy in question 4.2. You should explain why your algorithm uses no more than d colours.

   Let $G = (V, E)$ be graph such that it has maximum degree d and there exists $v \in V$ so that $\deg(v) < d$.

   We make a claim that if such a graph is connected, then there must exist some node that is connected to a degree d node such that it has degree less than d.

   If we assume that every node connected to a degree d node has degree greater than or equal to d then every node connected to it has degree d since we assumed maximum degree of d. But then we cannot have the node with degree less than d be connected to any of the degree d nodes, otherwise it would then have degree d, thus this graph must be disconnected. The contrapositive gives us the first claim.

   First observe if node v with $\deg(v) < d$ connected to the degree d node is removed from the graph, then its neighbours must have their degree decrease in the deleted graph. Including the degree d node.

Thus, we have guaranteed there will always be a node with degree less than d since we can decrease the degree of the degree d node, given the existence claim of a node connecting to it. If no such node exists then this implies that the maximum degree of the deleted graph is no longer maximum degree d.

In other words, continuous deletion of the node with degree less than d creates another sub-graph with the existence of such a node.

We then construct the following ordering, O:

```
1: procedure CREATE ORDERING(G = (V, E))
2:     Set O to empty ordering.
3:     while V is not empty do
4:         if there exists a degree d node in G then
5:             Find v ∈ V such that deg(v) < d and it is connected to a degree d
6:         else if there is no such degree d node in G then
7:             Find v ∈ V such that deg(v) < d
8:         end if
9:         Add v to the front of O
10:        Delete v from V
11:        Delete all e from E such that e is incident to v
12:    end while
13:    return the ordering
14: end procedure
```

Claim: Let $G = (V, E)$ be a connected graph with maximum degree d. Then the ordering produced from Create Ordering uses at most d colours when the greedy algorithm is run on it.

Proof: Let $O = (v_1, v_2, \ldots, v_n)$ be the ordering produced from the proposed algorithm.

We start with an empty graph $G' = (V', E')$ of empty edges and vertices. We will in order return the vertices and edges.

Consider the subgraph $G'(v_d)$ that contains all the nodes up to v_d in the ordering and the associated edges between those vertices.

Now, observe that v_d has at most $d - 1$ neighbours by construction of the ordering.

Then by removing v_d from this subgraph, the remaining $d - 1$ nodes clearly have at most $d - 1$ edges, and thus its maximum degree is $d - 1$. Hence, it will be can coloured with at most d colours through greedy. Since v_d has at most $d - 1$ neighbours, then it can still be coloured uniquely by the other d colours.

For each addition from v_{d+1} to v_n, by construction each of them had at most $d - 1$ neighbours at the time of removal and thus can be uniquely coloured from the other used d colours.

This proves the claim.

4 Weekly Meeting Logistics

You're the manager of an animal shelter, which is run by a few full-time staff members and a group of n volunteers. Each of the volunteers is scheduled to work one shift during the week. There are different jobs associated with these shifts (such as caring for the animals, interacting with visitors to the shelter, handling administrative tasks, etc.), but each shift is a single contiguous interval of time. Shifts cannot span more than one week (e.g., we cannot have a shift from 10 PM Saturday to 6 AM Sunday). There can be multiple shifts going on at once.

You'd like to arrange a weekly meeting with your staff and volunteers, but you have too many volunteers to be able to find a meeting time that works for everyone. Instead, you'd like to identify a suitable subset

of volunteers to instead attend the weekly meeting. You'd like for every volunteer to have a shift that overlaps (at least partially) with a volunteer who is attending the meeting. Your thinking is that you can't personally meet with every single volunteer, but you would like to at least meet with people who have been working with every volunteer (and may be able to let you know if a volunteer is disgruntled or having any difficulties with their performance, etc.). Because your volunteers are busy people, you want to accomplish this with the fewest possible volunteers.

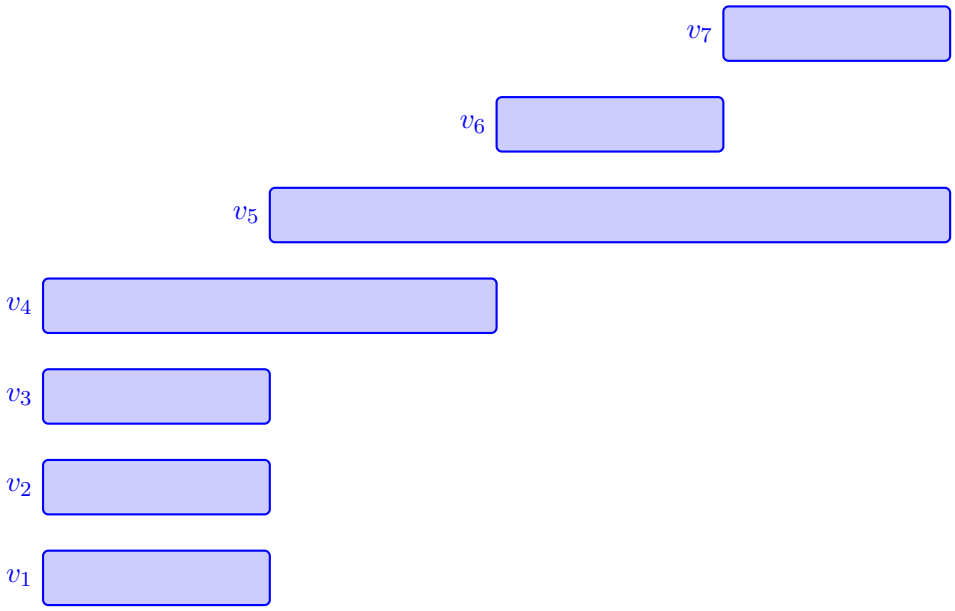
Your volunteer shifts are given as an input list V, and each volunteer shift v_i is defined by a start and finish time (s_i, f_i). Your goal is to find the smallest subset of volunteer shifts such that every shift in V overlaps with at least one of the chosen shifts. A shift $v_i = (1, 4)$ overlaps with the shift $v_j = (3, 5)$ but not with the shift $v_k = (4, 6)$.

1. [2 points] Your co-worker proposes the following greedy algorithm to select volunteers for your meeting:

   > Select the shift v that overlaps with the most other shifts, discard all shifts that overlap with v, and recurse on the remaining shifts.

   Give and briefly explain a counterexample to prove that this greedy strategy is not optimal.

   Consider the instance:

   We see that in the proposed greedy algorithm that v_4 intersects with the most amount of shifts, 4.

   After removing all intersections, only v_6, v_7 remains, since those two are disjoint, the greedy algorithm returns volunteers v_4, v_6, v_7 as its best solution.

   But notice that, by taking volunteers the two v_1, v_5 we sufficiently cover all shifts in the set, thus this greedy algorithm is not optimal on all instances.

2. [3 points] Give a greedy algorithm to solve this problem. Give an unambiguous specification of your algorithm using a **brief, plain English description**. Do not write pseudocode or worry about implementation details yet. (You may do this in part 5 if you feel that it's necessary to achieve a particular runtime.)

   Sort the shifts in terms of ascending order by finish time, for any ties, place first the earliest start time.

   Choose the first shift in the ordered list, and add it to our solution set.

   Then delete any intersecting shifts to that shift and remove that shift itself from the ordered list.

   Continue to the next shift in the ordered list and repeat the previous steps until the list is empty.

3. [2 points] In this and the next question, you will prove the correctness of your greedy algorithm. To start, give a "greedy stays ahead" lemma that you can use to prove that your greedy algorithm is optimal. You do not need to prove the lemma (that comes next). Your lemma should compare a partial greedy solution to a partial optimal solution and describe some way in which the partial greedy solution is "ahead."

   Denote greedy solution by $G = (g_1, g_2, \ldots, g_k)$. It is clear that every shift intersects with itself.

   Let $O = (o_1, o_2, \ldots, o_l)$ be optimal solution whose shifts are ordered in the by finish time as in G.

   Lemma: $f(g_i) \leq f(o_i)$, for each $i \leq \min(l, k)$.

4. [4 points] Prove the correctness of your algorithm. That is, prove that your set of selected shifts will (a) overlap with all the shifts in V, and (b) contains as few shifts as possible. You should do this by proving the greedy stays ahead lemma you stated in the previous question and using it to prove that the greedy solution is optimal.

   We prove the lemma by induction on i, where $i \leq \min(k, l)$, and $G = (g_1, g_2, \ldots, g_k)$, $O = (o_1, o_2, \ldots, o_l)$ are the greedy and optimal solutions respectively, both sorted in increasing order of finish time.

   Base Case ($i = 1$): The greedy algorithm selects the shift with the earliest finish time among all shifts. Since o_1 is a valid first choice in some optimal solution, the greedy algorithm must finish no later:

   $$f(g_1) \leq f(o_1)$$

   Inductive Hypothesis: Assume for some $i \geq 1$, we have:

   $$f(g_j) \leq f(o_j) \quad \text{for all } j \leq i$$

   Inductive Step ($i + 1$): We want to show that $f(g_{i+1}) \leq f(o_{i+1})$.

   Let t be the earliest time after which both greedy and optimal resume selecting shifts — that is, the time immediately after $f(g_i)$. Let S be the set of remaining unprocessed shifts that do not overlap with $g_1, \ldots, g_i$.

   By the inductive hypothesis, $f(g_i) \leq f(o_i)$, so any shift in O that does not overlap with $o_1, \ldots, o_i$ must also not overlap with $g_1, \ldots, g_i$ — i.e., it is in S. In particular, $o_{i+1} \in S$.

   Now the greedy algorithm chooses g_{i+1} as the shift in S with the **earliest finish time**. Since $o_{i+1} \in S$, we must have:

   $$f(g_{i+1}) \leq f(o_{i+1})$$

   Thus, the inductive step holds.

   Conclusion: By induction, we have shown that for all $i \leq \min(k, l)$, it holds that:

   $$f(g_i) \leq f(o_i)$$

5. [4 points] Briefly justify a good asymptotic bound on the runtime of your algorithm. If you prefer to present pseudo-code to help track the runtime incurred, you may do so.

   First, we sort the list V of n shifts in ascending order by their finish times. This takes $O(n \log n)$ time.

   Then we iterate through the list from left to right and start selecting shifts, once a shift has been selected we add it to the solution set. The first shift is always selected.

   Next, we traverse to find the next selected shift in the ordered list that does not intersect with it.

Intersection can be checked in $O(1)$ time, by verifying that the selected shifts finish time is after the start time of the current potential candidate shift we are looking at.

Notice in this process, each shift is visited at most once, either it is selected and added to the solution, or it is skipped due to intersection. Thus, the total traversal takes $O(n)$ time.

Combining both steps, the overall runtime of the algorithm is: $O(n \log n + n) = O(n \log n)$.