Homework 3: Greedy

Be sure to read the grading guidelines and style guidelines. Especially to see the suggested format for describing algorithms.

We sometimes describe how long are justifications or proofs are. These lengths are intended to help you estimate how much detail we're expecting, you should not take those estimates as hard length-limitations.

Our solutions for any individual problem will fit in approximately one page or less. **Beginning with this homework, if you submit an answer substantially longer than 2 pages, the TAs are allowed to stop reading at the end of page 2.**

You are allowed (and encouraged!!) to collaborate with each other. Brainstorming is much easier to do in a group than alone! But you must follow the collaboration policy (which includes needing to write your submission on your own).

You will submit to gradescope; we will have a different box for each problem, so please give yourself extra time to submit.

1. Reduction! [10 points]

For a weighted graph $G = (V, E)$, we call a set $S \subseteq E$ a "maximum spanning tree" if it is a spanning tree, and the sum of its edges is at least as large as the sum of any other spanning tree.

You are given the following library function.

```
MinSpanningTree
Input: A weighted graph made up of a set of edges E and a set of vertices V where the $i^{\text{th}}$ edge has weight $w_i$
Output: A minimum-weight set of edges such that you can get from any vertex of G to any other on only those edges
```

Explain how to use this library function to find the maximum spanning tree of a graph G, and briefly explain why it works (our full answer is only 3-4 setences).

2. Find a Counter-Example [10 points]

For a graph $G = (V, E)$, we call a set $S \subseteq V$ a "vertex cover" if $\forall (u, v) \in E : (u \in S \vee v \in S)$. In English, a vertex cover is a subset of the vertices so that every edge has at least one of its endpoints in the set.[1] We are usually interested in small vertex covers (since V is always a vertex cover). Consider the following greedy algorithm for finding a small vertex cover:

```
function GreedyVertexCover($G = (V, E)$)
    $S \leftarrow \varnothing$
    while $E$ is not empty do
        Let $u$ be the highest degree vertex                    ▷ not counting deleted edges
        Add $u$ to $S$
        Delete all edges incident to $u$ from $E$    ▷ we know these are covered by $u$ and can ignore them
    return $S$
```

Prove this algorithm is incorrect by counterexample. Specifically, give an example of a graph on which this algorithm does not produce the smallest vertex cover. Show both the vertex cover the algorithm finds and a smaller cover. You may not use the graph discussed in lecture 8 to show the other greedy VC algorithm not optimal.

[1]The name is somewhat counter-intuitive; the set contains vertices – we say "the vertices cover the edges" so it's the **edges** that are covered (by the *vertices*) in a *vertex* cover.

3. Just A Proof [25 points]

In this problem you'll learn a different greedy algorithm for MSTs and prove it correct.

Let $G = (V, E)$ be a graph where edge i has weight w_i. Call the edge e "redundant" if removing e would not cause the graph to be disconnected.

Consider the following greedy algorithm

```
function RedundancyRemover($G = (V, E)$)
    Sort edges in decreasing order of edge weight.
    for each edge $e$ in decreasing order do
        if $e$ is redundant then
            delete $e$
    return remaining edges, which are an MST.
```

(a) Prove that if G is a connected graph, then `RedundancyRemover` produces a spanning tree (don't worry about minimum yet!) [5 points]

(b) Prove that if G is a connected graph with distinct edge weights, then `RedundancyRemover` produces the minimum spanning tree.
You may use as a fact that since the edge weights are distinct, the MST is unique. [20 points]
Hint: We strongly recommend using a structural result here. You should think about the structural result from class and see what the corresponding one here might be.

4. TAs need Thai food [25 points]

At the midterm grading party, TAs need to eat. Therefore, Robbie needs to provide food[2]. To order the proper amount, Robbie needs to know (in minutes) how long it will take to finish grading. And, ideally, he'd like grading to go as fast as possible (the less food ordered, the fancier it can be for the same price).

He knows for his group of the k TAs, that the i^{th} TA takes exactly t_i minutes to fully grade a single midterm (regardless of who submitted that midterm). Because he's only worried about food ordering for now, Robbie just needs to figure out the smallest number of minutes it would take to get the midterms graded he doesn't need to record how many midterms each TA grades.

(a) Design for Robbie an algorithm that runs in polynomial time $k, n, \max_i t_i$ which gives us the smallest number of minutes it takes to grade all midterms. For example, if we have variables $k = 3$ TAs, $n = 4$ midterms, and $t_1 = 1, t_2 = 3, t_3 = 60$ - then you should return 3.

(Hint: We only care about the number of minutes here, not which TA grades what. Your algorithm only needs to return the *number of minutes* the optimal configuration would take, **NOT** the optimal configuration of which TAs grade which midterms. This should make the pseudocode easier.)

(b) Prove that your algorithm gives exactly the optimal time in minutes.

(c) What is the running time of your algorithm? Briefly (3-4 sentences) explain.

5. Interval Scheduling Correction [25 points]

Robbie said in Wednesday's lecture that "fewest overlaps" is a correct greedy algorithm for interval scheduling.

Robbie was wrong! There are instances where "fewest overlaps" gives a non-optimal answer. You're going to find out why in this problem.

[2]Hangry TAs don't make for good graders

More formally, define the fewest overlaps heuristic as follows:

```
function FewestOverlaps(Intervals[])
    Solution ← empty list
    while Intervals is not empty do
        for each interval $u$ in Intervals do
            Count the number of intervals in Intervals overlapping $u$
        Let $v$ be an interval with the smallest number of overlaps        ▷ Break ties arbitrarily
        Add $v$ to Solution
        Delete every interval that overlaps with $v$.
    return Solution
```

Robbie thought there was a proof of correctness! He had something like this in mind.

Recall the Lemma from lecture

Lemma 1. *If u is the interval with the earliest end time, and u overlaps with v and w, then v and w overlap each other.*

Spoof. Since u has the earliest end time, anything it overlaps with must overlap with the last instant at which u is active (the "right endpoint" of u). Thus both v and w are active at u's right endpoint, but that means they overlap with each other, as claimed. □

We'll use a structural result here. The specific claim is that: for every k, after both `FewestOverlaps` and `EarliestEndTime` have selected their first k intervals, the remaining sets of valid remaining intervals (i.e., those not overlapping any of the k selected intervals) are the same.

Spoof. Suppose, for the sake of contradiction, the claim is false. We consider the first selection of an interval which would lead to different remaining sets of valid intervals.

Let u be interval with the earliest end time and this step of the algorithm, and suppose it overlaps with a other intervals.

Case 1: u is also selected by `FewestOverlaps`
This case is impossible — both algorithms select u, so they eliminate exactly the overlapping intervals, and since this was to be the first step of disagreement, we had eliminated the same sets until now. Thus they still have the same set of valid remaining intervals.

Case 2: u is not selected by `FewestOverlaps`
In this case, we'll use Lemma 1 to derive a contradiction. Call the intervals that u overlaps with $v_1, v_2, ..., v_a$; without loss of generality, let v_1 be the interval selected by `FewestOverlaps`. By Lemma 1, v_1 overlaps with each of $v_2, ..., v_a$, and it also overlaps with u. That gives us a list of a intervals overlapping with v_1.

Thus both `FewestOverlaps` and `EarliestEndTime` will eliminate all of $u, v_1, ..., v_k$ from their valid set of intervals. And by assumption, they started this iteration with the same sets of valid intervals. But we assumed this was the first step at which the algorithms would eliminate different sets. We have defined the v_i as the only intervals overlapping with u, so it must be that v_1 also intersects an additional interval which will be removed (or else the valid sets remaining would be the same). But then v_1 overlapped with at least $a + 1$ intervals. This contradicts the definition of `FewestOverlaps`, as it would not select v_1, which had more overlaps than u. □

(a) Fewest overlaps doesn't actually "pass the smell test" for an algorithm idea — adding in a bunch of copies of an interval could change how many elements it overlaps with, but won't change the optimal answer.
Design a counter-example to the optimality of `FewestOverlaps`. Drawing a picture is probably the easiest way to communicate a counter-example. Also include: which elements are in the optimal solution, and which elements `FewestOverlaps` will find. [10 points]

(b) The proof must be wrong! Clearly describe the error. A description cannot just be "the counter-example shows it must be wrong." You need to find an incorrect leap in logic, invalid assumption, missing case, or similar error. Nonetheless, it will probably help to run the algorithm on your example with an eye toward the proof to see where it breaks down.
Our explanation is only a few sentences. [15 points]