Introduction to Algorithms	Problem Set 2 (3 problems)
CS 4820 Fall 2019	Solutions

(1) Implementing Kruskal. In this problem we consider the classical Kruskal's minimum-cost spanning tree algorithm (algorithm A from the lecture on Monday, September 9th). Given n nodes V, and m possible edges E, each $e \in E$ with its cost $c_e \geq 0$. **Implement the algorithm till there are 3 components left**. The output of this algorithm will be a set of $n-3$ edges that give a graph on V with 3 components. **Your algorithm should output the sizes of the three component in non-decreasing order.**

Your algorithm needs to run in $O(m \log m)$ time. To do this, it is helpful to keep an array called componentName where componentName$[v]$ for a node v is the name of component containing node v. Then, testing if an edge (v, w) connects nodes in different components is simply a test of whether componentName$[v]$ $\neq$componentName$[w]$. The hard part is to make sure you can update the componentName array as components merge. The main idea is that when two components merge, the merged component should inherit the name from the larger of the two, this way any node v will change its component name componentName$[v]$ at most $\log_2 n$ times, as each change increases the value by at least a factor of 2. See page claim (4.23) on 153 of the book.

Solution Idea: Using the above idea we have an array called componentName, where componentName$[v]$ is a name for the connected component of node v. Maybe the most non-trivial part of this implementation is how will the algorithm know which of the two sides is bigger in $O(1)$ time. To facilitate this, there are a couple of options. We can just keep a linked list of nodes in each component. To decide if an edge $e = (v, w)$ should be added to the tree we check if componentName$[v]$ =componentName$[w]$. If we do decide to add the edge, we need to know if v or w has the smaller component: we can do this by walking down simultaneously on both component's linked list of members, and the smaller is the one that ends first. Now we need to merge the component, which requires updating componentName on the smaller side, and one list to the other one.

Alternately to walking down both lists to see which is shorter, we can also keep an explicit count for the size componentSize. However, we cannot afford to keep this updated for all nodes, as the size changes also for the bigger side of when we merge components. Instead, we can just update it at the node that is the name of the component. So then checking which side is bigger would be comparing componentSize[componentName$[v]$] and componentSize[componentName$[w]$], and updating the value to componentSize[componentName$[v]$+]componentSize[componentName$[v]$] at the one that has the bigger size (and hence is the new name).

(2) Pairing Students. You are helping a group of students get involved in open source coding. The plan is to have pairs of students spend some time t each week coding for the project together. We will assume that you have n students, and every student prefers to work in a pair, if possible. To facilitate the process of organizing these pairs, you've asked each student i to specify a single time interval $I_i = [s_i, f_i]$ that available to them every week. We call a pair (i, j) of two students an *eligible pair* if their intervals of time overlap by at least t. Give an algorithm that selects as many eligible pairs as possible such that every paired student is part of at most one of the selected pairs, and runs in time at most $O(n^2)$.

Solution: Select the interval that ends first. If this doesn't overlap with any other intervals, there's clearly no way to pair it. Hence, consider the interval that ends first and overlaps with some other intervals. Pair it with an overlapping interval that ends as early as possible. Now recurse on the remaining intervals.

Alternate Algorithm: it is equally good to start from the end: sort the intervals by decreasing start time, select the person who starts last, and pair him/her with the person intersecting with him/her that starts as late as possible.

Running time is again easily $O(n^2)$, and probably better.

Correctness by exchange argument. Let M be our matching and M^* some other optimal matching. We want to prove that M is also optimal by exchanging M^* to make it more and more like our matching, while keeping it optimal along the way. Consider the intervals sorted by end time, as we did in the algorithm. And let interval $[s_i, t_i]$ be the first that M and M^* differ on. There are a bunch of cases

- Suppose i is unmatched in M. Then given that in all previous intervals M and M^* agreed, there is nothing i can be matched with in either schedule, so M and M^* must agree on i.

- Suppose i is matched to a person j in M, but unmatched in M^*. We can modify M^* by matching (i, j) and possibly leaving j's partner in M^* unmatched. This new matching is the same size, so it is also optimal.

- Suppose i is matched to a person j in M, and is also matched in M^* to a different person k. This case needs to break further: depending on if j is matched in M^* also. If that is not the case, we can modify M^* again: match (i, j) and leave k unmatched. Again, this results in the same size matching, so it is also optimal.

- The final case is shown on the figure below: Suppose i is matched to a person j in M, and is also matched in M^* to a different person k, and j is matched to yet another person m in M^*. This is depicted on the figure below, where the matching M^* matched the two red intervals, and our algorithm matched (i, j), so the interval for j must be the earliest that ends among those starting during i's interval, as shown in the figure. We claim that now we can modify M^* by matching

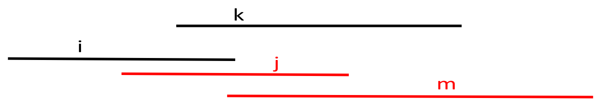

Figure 1: Last case of the exchange argument.

(i, j) and (m, k). This is clearly the same size matching. What we need to prove is that m and k overlap by at least t. Note that j ends before k does, as our algorithm selected j. Since m's interval started in time to overlap with j for t time (as it was matched with j in M^*), it overlaps with k at least that much.

Common mistakes:

A greedy algorithm that doesn't work: Maybe the most common non-working algorithm is to select the person that overlaps with the fewest people, and match him with the neighbor that has fewest neighbors. In the figure next you select red, and then match it to the blue, and then two will remain unmatched. But there is a solution matching all.
Dynamic programming: Some students are trying is to sort the students by finish time $f_1 \leq \ldots, f_n$, and use dynamic programming with subproblems where $Opt(i)$ is the optimal solution for students

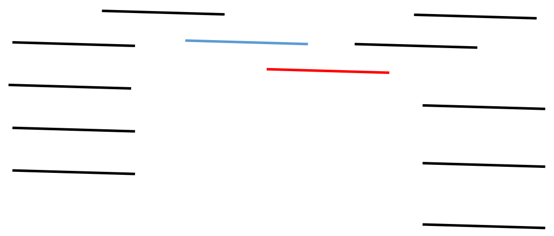

Figure 2: Example for wrong greedy.

$1, \ldots, n$. A common way to try a recurrence is to check if student i can be matched with $i-1$, and then use subproblems $Opt(i-1)$ or $opt(i-2)$. This solution can only match student i to either $i-1$ or $i+1$. However, it may easily be better to match student i with a different other student (especially if $i+1$ and $i-1$ have long intervals, and hence overlap with many more students.

Mistakes in the proof For students with the optimal greedy algorithm, some tried use a ”greedy stays ahead” style proof. We are not aware of a possible claim that greedy can stay ahead. To show a greedy stay ahead proof one needs a mathematical statement of the form that after i steps the greedy is no worse in a well-defined way than the first i matched of an optimal solution. We are not aware of claims of this form that may be true.

The hard part of the exchange argument is the final case. Some students are missing a good argument for that case.

(3) Shift Scheduling. Consider the following problem where Alice is considering shifts to work. She wants to plan a sequence of n shifts, labeled as shifts $i = 1, \ldots, n$. For each shift i, if she works that shift, she gets a known rewards r_i. However, by law (and common sense) she cannon work two consecutive periods, that is, needs to take a break between any two shifts. To illustrate the problem, consider the following example. There are 5 shifts, and the rewards in order are $8; 6; 4; 9; 2$. So Alice can work shifts 1,3,5 and earn 8+4+2, a total of 14 reward. But she can do better, if she works shifts 1 and 4 only, earning a total reward of 8+9=17.

Give a linear time algorithm that finds the maximum reward that Alice can achieve.

Solution: Set up a dynamic program where $Opt[i]$ is the optimal solution for shifts $1, \ldots, i$. For the base case we have $Opt[0] = 0$ and $Opt[1] = r_1$.

For recurrence we have $Opt[i] = \max(Opt[i-1], r_i + Opt[i-2])$.

Using this recurrence we can compute all $Opt[i]$ values in order $i = 2, 3, \ldots, n$.

Now the optimum value we need to return is $Opt[n]$.

Running time is easily $O(n)$: there are $n+1$ values to compute, and computing each involves one addition and one comparison.

Solution backwards Naturally, one could do this all backwards with $\bar{O}pt[i]$ = the optimal solution for shifts $i, \ldots, n$. In this case the base case is $\bar{O}pt[n+1] = 0$ and $\bar{O}pt[n] = r_n$. And the recurrence is $\bar{O}pt[i] = \max(Opt[i+1], r_i + Opt[i+2])$. Now the optimal value is $\bar{O}pt[1]$.

Proof of correctness for dynamic programming A correct proof for a Dynamic Prgramming algorithm consist of the following steps:

- Argue that the base case is correct (trivially true in this case by definition)
- Show that for all i we have $Opt[i]$ correctly computed: To see why Alice either works on day i or she does not. If she does not work on day i, then the maximum she can get is $opt[i-1]$ by the

induction hypothesis. If she does work on day i, that has reward r_i, and then she cannot work on day $i-1$, so by the induction hypothesis she can get $Opt[i-1]$.
- Argue that the correct value is returned. (Again trivial in this case).

Solution via reduction . We can solve the problem by reducing it to weighted interval scheduling. For any day i, associate an interval $[i-0.6, i+0.6]$ (so that overlapping intervals represent conflicting shifts) and weight $w_i = r_i$. Now solve the weighted interval scheduling problem and return the resulting optimal value. Running time is that of weighted interval scheduling, which is $O(n \log n)$ unless you notice and explain why this special case can be implemented faster (which is true due to start and end times sort in the same order).

For proof of correctness we need to explain why this weighted interval scheduling problem is equivalent. This is true as neighboring intervals overlap, and others do not. So we can select a set of intervals if and only if the selected set is a set of days Alice can work.

Common mistakes Maybe the most common mistake is to only have one base case in the dynamic programming solution: sone students used $Opt[1] = r_1$ only as base case. With this base case, the recurrence doesn't work for $i = 2$ as it used $i-2$, which is not defined.

Some students didn't define the meaning of $Opt[i]$ of said something like "optimum solution for problem i" without saying what "problem i" is. Recall from the description on the top of the homework that this is required for full credit!

A few students tried to use a greedy algorithm. We are not aware of any reasonable greedy algorithm for this problem.