CS 330, Fall 2020, Homework 10
Due Wednesday, November 18, 2020, 11:59 pm Eastern Time

1. **Cell tower problem**

   The company operating the cell towers is at it again. This time they need to install towers along a long country road connecting two cities. Help them find the best location for the towers! Each installed tower covers an area of 5 miles' radius (i.e. 5 miles on either side of the tower). There needs to be cell coverage along the entire road. There is infrastructure to install a tower at every milestone along the road. The total distance between the cities is n miles. (You are allowed to place a tower in either city.) The cost of installing towers varies by location. The costs of the different locations are given to you in an array `cost[ ]` of the length $n + 1$. (The costs are positive numbers.) You need to find a set of building sites such that the entire road has cell coverage and the total cost is minimized.

   (a) Observe that it is possible that the minimum cost solution consists of stretches of road that are covered by multiple towers. Prove however, that in any *optimal* solution, each stretch is covered by at most 2 towers.

   (b) Consider a greedy algorithm where we add locations to our solution set one at a time. We consider the stretches of road (the road between two neighboring tower locations) in increasing order of distance from the first city. Each time we encounter a so far uncovered road stretch r_i (that is between miles $i-1$ to i), we will add a tower location j that (1.) covers r_i (2.) has the best cost-coverage ratio. That is, we choose a location j such that the relative cost $\frac{\texttt{cost[j]}}{\texttt{additional miles}}$ of every additional mile covered by this tower is minimized.

   Show an example where this algorithm fails.

   (c) Here is the pseudocode for a memoized algorithm to find the *minimum cost* of locations. (Note that it does not give you the actual set of locations.)

   Algorithm 1: DPMinCostTowers(`cost`, n)

   ```
   1 M ← empty array of length n;
   2 for i = 0 ... n do
   3     M[i] ← {some appropriate formula to be defined by you};
   4 return M[n];
   ```

   What does the entry `M[i]` contain (that is, state the subproblem for which it provides a solution)?

   Write the recursive formula to compute `M[i]` in line 3 of the algorithm. (Write the formula only, no need for explanation.)

   (d) Write an algorithm that takes the table M filled in by Algorithm 1 and returns a list of the locations where towers should be built.

2. **Game tournament**

   You are participating in an online gaming tournament. The tournament consists of n competitions. You are eligible to participate in any of them. However, you can only be in one

competition at a time, there can be no overlap between the tournaments you enter. Sadly, you'll have to make a choice in which competitions you participate. As an input to this problem you will be given an array `start[ ]` of length n containing the start time of each competition, and another array `duration[ ]` with the duration of each game. You may assume that the games are indexed 0 through $n-1$ in increasing order of their start times.

Since you play for the pure enjoyment of the game, you don't care about your placement in any of the competitions. Your goal is to pick games so that you **maximize** the time that you spend playing. (If you start a game, you have to finish it.)

(a) What algorithm from class could be used to solve this problem? How would you transform the inputs to this problem into the ones needed for the algorithm from class? Make sure you understand how that algorithm works before proceeding.

(b) Due to the large number of entries, the tournament organizers have decided to limit the number of competitions each player can enter to k (so the input now consists of the arrays `start` and `duration`, and the number k). Your goal is to find a set of at most k tournaments that do not overlap, and whose total duration is as long as possible. Design a polynomial-time dynamic programming algorithm for the problem. Please organize your answer as follows:

i. Define the set of subproblems that your algorithm will solve.
[*Hint:* Consider having one subproblem for each $i \in \{1, ..., n\}$ and $j \in \{1, ..., k\}$. Think carefully about the order in which to consider the tournaments.]

ii. How many such subproblems are there?

iii. Suppose we create a table M to store the value of the subproblems we solve. Write a recursive formula to compute `M[i,j]` in terms of other entries of M (that is, the solutions to smaller subproblems). (Be mindful to include the base cases.)

iv. Explain briefly why your formula is correct. (This explanation is really the proof of correctness for your algorithm.)

v. Write pseudocode for your algorithm (it should fill the memoization table and find the optimal subset of tournaments).

vi. What is the asymptotic running time of your algorithm? (Only write the Θ formula, no proof needed.)

To give you an example to follow, here are the answers we might have given (except for the pseudocode) if the question were about the Knapsack algorithm from class:

i. The subproblems: For each $i \in \{0, 1, 2, ..., n\}$ and $w \in \{1, ..., W, \}$: compute $OPT(i, w)$, the maximum value one can obtain by using only items $\{1, ..., i\}$ and a knapsack with capacity w.

ii. $(n+1) \cdot W$ (since there are $n+1$ choices for i and W choices for w).

iii. When $i = 0$, there are no items being considered, so the maximum value you can get is 0. When $i \geq 1$, there are two possibilities: either the optimal solution uses item i, or it doesn't. Among solutions that don't use item i, the best one will have value $OPT(i-1, w)$ (since any solution among items $\{1, ..., i-1\}$ that weights at most w kg is feasible). Among solutions that do use item i, the best one will have value $value_i + OPT(i-1, w-weight_i)$, since I can combine item i with any solution

that weights at most $w - w_i$ kg. The formula takes the better of those two solutions (or just the one without item i if item i is over the weight limit).

iv.

$$OPT(i, w) = \begin{cases} 0 & \text{if } i = 0, \\ OPT(i-1, w) & \text{if } weight_i < w, \\ \max(OPT(i-1, w), value_i + OPT(i-1, w - weight_i)) & \text{otherwise.} \end{cases}$$

v. We won't reprint the pseudocode here. See the book, the lecture slides, or the supplied Python code for pseudocode (and working code!) for the Knapsack algorithm.

vi. $\Theta(nW)$ time. Justification: Each entry of the table takes $O(1)$ time to fill and there are $\Theta(nW)$ entries. Computing the optimal solution from the table takes $O(n)$ time. The total is $\Theta(nW)$.