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

Contributors: Erin McAteer

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 4 and returns a list of the locations where towers should be built.

ANSWER

a) If we assume that the minimum cost solution can consist of stretches of road covered by multiple towers, then when computing the optimal solution, ie: the solution with the smallest amount of towers, at the smallest cost that covers the entire road, we can, and likely will encounter locations where two towers overlap.

Now let's assume that at some stretch of road, we have 3 overlapping towers. This can only really happen when one of those towers is covering an area that is already completely covered by the other two. This means that, if we only have two towers overlapping partially, and we add another tower that overlaps at the same time as the other two (causing a triple overlap), it will, by design, either be placed in the same milestone as the first tower, in the same milestone as the third tower, or somewhere in between, which means whatever area it covers, is already covered, thus it cannot be a part of the optimal solution.

If it isn't placed in one of those three locations, then it would never cause a triple overlap.

b) If while searching for a location j that covers r_i the algorithm finds two locations j_1 and j_2, where $relativeCost(j_1) = 2$ and $relativeCost(j_2) = 3$, but if we pick j_2 we cover r_i, but leave r_{i+1} uncovered, while if we pick j_2 we cover both r_i and r_{i+1}, yet the algorithm will go for the lowest relative cost position j_1. However, we then encounter r_{1+i}, and here we are faced again with a choice between j_2, and a new position j_3, let us now assume that j_3 has the least *relativeCost* when compared to j_2, well this could lead to a situation where in total, we increment more costs since we have to pick two towers when a single one would do, since we can't recurse back and pick another one. This could happen if both j_1 and j_3 had really low cost when and cover an average amount of additional miles, and j_2 had an average cost and a more well spread out aditional miles, but still not enough to break the *relativeCost* requirements.

c) $M[i]$ is providing the minimum cost to cover up to location n

Algorithm 2: DPMinCostTowers(`cost`, n)

1 $M \leftarrow$ empty array of length n;
2 **for** $i = 0 \dots n$ **do**
3 $\quad M[i] \leftarrow \{$if $i = 0$ then $cost[0]$ else if $1 \leq i \leq 5$ $min(cost[i-1], cost[i])$ else if $6 \leq i \leq 8$
$\quad\quad min(cost[i-1], cost[i] + cost[i-6])$ else
$\quad\quad min(cost[i-1] - cost[i-8], cost[i] + cost[i-6])\}$;
4 **return** $M[n]$;

d)

Algorithm 3: DPTowerList(`M`, `costs`, n)

```
1  TowerList ← empty LinkedList;
2  lastAdded = −1;
3  cost ← 0;
4  for i = 0 . . . n do
5      if M[i] > cost then
6          if M[i] = cost + cost[i] then
7              TowerList.add(i);
8              cost ← M[i] lastAdded ← i;
9          else
10             for j = max(i − 5, 0) . . . i do
11                 if M[i] = cost − cost[lastAdded] + cost[j] then
12                     TowerList.remove(lastAdded);
13                     TowerList.add(j);
14                     cost ← M[i] lastAdded ← j;
15     else if M[i] < cost then
16         TowerList.remove(lastAdded);
17         TowerList.add(i);
18         lastAdded ← i;
19         cost ← M[i];
20 return TowerList;
```

The runtime of this algorithm is $O(n)$. My reasoning behind it is quite simple, given the finalized Minimum Cost Array M, and the last mile marker we want to cover until n, and the costs array *costs*, what the algorithm will initialize a variable *cost* as 0, and a *lastAdded* as -1. From here it will iterate through the Minimum Cost Array M, and every time an element of the array $M[i]$ is larger than *cost* (which is the current minimum cost up until position $i-1$), it'll first check if the cost difference between the last known point and $M[i]$ is the same as $cost[i]$, if so we found our tower and we can add it, if not, then it'll preform a sweep check to see which of the other towers in the range of i does the equation add up to, and remove the last tower and add this new one.

This works because the Minimum Cost for an element i in M only increases whenever we place a tower, or replace the last one we placed, and so we can monitor for that increase and add the respective tower to the *TowerList*. If however we see a decrease, we know our last tower was removed in favor of tower i, so we can preform the removal/addition required.

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. (For example, for the Knapsack problem, the subproblems were: "for each $i \in \{1, ..., n\}$ and $w \in \{1, ..., W, \}$: compute the maximum value one can obtain by using only items 1 to i and a knapsack with capacity w.")
[*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.)

ANSWER

a) We could use a Knapsack style algorithm to solve this problem. For each item (which would represent a particular competition), we'd give it a value of it's start time, and a weight of it's duration. Furthermore, we'd set the total W to be the total time we have to play in the tournament, and n would be the number of total games that are available to us.

b)

i. The subproblems we need to solve are, for each pair of start/duration $i \in \{1, ..., n\}$, and for each possible amount of games played $j \in \{1, ..., k\}$, what is the biggest amount of time I can possibly spend playing in a competition, while only using pairs 1 to i, and a maximum of games played j.

ii. This would result in $n \times k$ different subproblems.

iii. In order to solve this, I'm gonna need a secondary array $p[]$, that for a specific competition indexed at i, will return the last competition to finish j, before i began. They can get this by adding up each of the start and duration values for every competition indexed as $l < i$, prior to running the actual formula below we'd also assume that $p[1] = 0$.

$$M[i][j] = \begin{cases} 0 & \text{if } k = 0 \\ 0 & \text{if } i = 0 \\ max((duration[i] + M[p[i]][k-1]), (M[i-1][k])) & \text{for all other cases} \end{cases}$$

iv. In order to understand this algorithm, we need to focus ourselves on the optimized set O, then at each iteration, we have two options, either activity i is in the set, or it isn't. If it isn't then the set at i, for a specific $j \in \{1, ..., k\}$ will be equal to the set at $i - 1$ (or at the activity that came before it) for the same j.

If however, activity i is in the optimized set, then that set includes i and the last activity to finish before i, or $p(i)$.

v.

Algorithm 4: maximumGameTime(*start*[], *duration*[], k)

```
1  n ← start.length();
2  p ← empty array of length n;
3  for i = 1...n do
4      for j = i − 1...0 do
5          if j == 0 then
6              p[i] = 0;
7              break;
8          if start[j] + duration[j] ≤ start[i] then
9              p[i] = j;
10             break;
11 M ← empty 2d array of length n × k;
12 for i = 0...k do
13     M[0][i] = 0;
14 for i = 0...n do
15     M[i][0] = 0;
16 for i = 1...n do
17     for j = 1...k do
18         M[i][j] = max(duration[i] + M[p[i]][k − 1], M[i − 1][k]);
19 return M[n, k];
```

vi. My algorithm, as built above, will run in $\Theta(n^2)$, due to the construction of p. Although should be noted that the remaining of the algorithm will take $\Theta(n \times k)$.