Name: Kevina Wong

ID: 109179049

CSCI 3104, Algorithms **Profs. Chen & Grochow**

Problem Set 9 – Due Fri Apr 10 11:55pm **Spring 2020, CU-Boulder**

Advice 1: For every problem in this class, you must justify your answer: show how you arrived at it and why it is correct. If there are assumptions you need to make along the way, state those clearly.

Advice 2: Informal reasoning is typically insufficient for full credit. Instead, write a logical argument, in the style of a mathematical proof.

Instructions for submitting your solutions:

- All submissions must be typed.
- You should submit your work through the **class Canvas page** only.
- You may not need a full page for your solutions; pagebreaks are there to help Gradescope automatically find where each problem is. Even if you do not attempt every problem, please allot at least as many pages per problem (or subproblem) as are allotted in this template.

Quicklinks: 1a 1b 1c 2a 2b

Name: Kevina Wong
ID: 109179049

CSCI 3104, Algorithms **Profs. Chen & Grochow**
Problem Set 9 – Due Fri Apr 10 11:55pm **Spring 2020, CU-Boulder**

1. Could dynamic programming be applied to give an efficient solution to the following problems? Justify your answer in terms of the optimal substructure property on the overlapping sub-problems. Even if you think the answer is yes, you do not need to give an algorithm; the question is *not* "show us how to solve it using DP", it is "give an argument as to whether DP is a reasonable approach to try here."

   (a) List maximum.
   Input: List L of numbers.
   Output: The maximum element in the list.
   Answer: The *Optimal Substructure Property* says that the value of an optimal solution can be computed in terms of overlapping sub-problems. Basically, the way that dynamic programming works is that it can break up a problem into overlapping sub-problems with the same properties as the original problem, then it solves the sub-problems and memorizes the solutions to be saved for future use. Then, it can build solutions to larger and larger sub problems.
   While dynamic programming can be applied to this problem, it doesn't yield an efficient solution. This problem can be done recursively and applied using dynamic programming. Our sub-problems can be the max element in sub-lists L[0:1], L[0:2]...L[0:len(L)-1]. Within each of these sub-problems, there are overlapping sub-problems. To elaborate, when we want to find the max element in the list L[0:i], our overlapping sub-problem would be the solution to the list L[0:i-1]. Because it is overlapping, we can pull a memoized solution from a table/list rather than recomputing the max element. With this, the optimal substructure property holds, and there are overlapping sub-problems. However, this process would not be efficient. Dynamic programming takes advantage of the fact that there are overlapping problems and memoizes it to reduce the time and space needed to recompute solutions that were already found before. However, in this case, each sub-problem would only overlap once rather than many times. For example, when we want to find the solution in L[0:3]. We previously would've solved our first sub-problem of finding the max-element in L[0:1] and memoized it. Then, we would've solved sub-list L[0:2] by pulling the memoized solution from L[0:1] and compare it with L[2]. Then, we would use the memoized solution of L[0:2] to find L[0:3] instead of recomputing L[0:2]. However, after being used once, the sub-problem L[0:1] becomes useless and will never be useful again. Along those same lines, when we begin to compute L[0:4], L[0:2] becomes useless. Because it's unnecessary to re-access the solutions of the sub-problems more than once in this case. Dynamic programming isn't actually efficient here, which is why while it can be applied, it'd be faster to stick with a linear search as we don't have to

waste time and space to put the solutions of the sub-problems into a memoized list.

(b) Rod cutting.

Input: A list of values $v_1, \ldots, v_n$ for rods of length $1, \ldots, n$, respectively.

Goal: Divide a rod of length n into pieces of lengths $\ell_1, \ldots, \ell_k$ (k can vary) to maximize the total value $\sum_{i=1}^{k} v_{\ell_i}$.

Note: While this problem is discussed on GeeksForGeeks (and elsewhere), the explanation of optimal substructure on GeeksForGeeks, while not incorrect, is not sufficient explanation to demonstrate mastery of this question.

Answer:

Recall that for the Optimal substructure property to hold, the problem must be able to be split up into smaller sub-problems, and the sub-problems must be overlapping. This means that to the solution to the smaller subproblem is reused over an over to solve a larger sub-problem.

Therefore, since an optimal solution can be made up of the solutions to this problem, we can say that dynamic programming can be applied to find an efficient solution to the rod cutting problem. The optimal subproblem for this problem can be expressed in the following way.The rod of value n can be cut as rods of length $l_{0:m} + l_{m:n}$, with $m \in \mathbb{N}$. Then, the sub-problem would be to find the max value of $l_{m:n}$, and can be solved by recursively calling this process and returning the max value each time. The idea behind this is that by recursively calling this process, we would be getting the maximum value each time for a smaller and smaller length. When finding the max value for a given length, a max value of a length might have to be used several times. Rather than recomputing it, it can just find it in a memoized table/list instead of needing to repeatedly compute the same value. For example, if we wanted to find the max value of a rod of length 7, we might need to compute which rod splits have the best value between splitting it as rods of length 4, 3; 4, 2, 1; or 4, 1, 1, 1. As you can see in this example, 4 would be the "overlapping subproblem". Instead of needing to recompute it three times, dynamic programming can just take the memoized max value of length 4. Many times in any rod splitting problem, an overlapping sub-problem will need to be used, thus making these sub-problems overlapping which makes it useful for us to memoize solutions to sub-problems. Therefore, because the optimal substructure property holds and because there are overlapping subproblems, dynamic programming can be applied to this problem.

(c) Graph 3-coloring.

Input: A simple, undirected graph G.

Goal: Decide whether one can assign the colors $\{R, G, B\}$ to the vertices of G in such a way that no two neighbors get the same color.

Hint: You may find the notion of critical graph useful.

Answer:

While this problem can be broken down into sub-graphs, dynamic programming cannot be applied to this problem. Since we want to do a 3-color graph coloring, we can ultimately conclude that the smallest sub-graph to this problem is a critical graph with chromatic number 3, consisting of three nodes and three edges. This is because this is the smallest our main problem can be broken down while maintaining the same properties as the original problem. The larger sub-graphs can be made up of combinations of our original sub-graphs thus showing that the sub-problems are overlapping. However the reason why dynamic programming fails in this problem is that when the sub-graphs are recombined it loses the properties of the original problem as it cannot successfully ensure that (a) no two neighbors are the same color and (b) the graph can actually be filled with only 3 colors within the recombination process. There is nothing that we can change about our sub-problem to ensure this, which makes our sub-problem useless in finding an optimal solution. Therefore, the optimal substructure property fails, and dynamic programming cannot be applied to solve this problem.

Name: Kevina Wong
ID: 109179049

2. Write down the recurrence for the optimal solution for each of the following problems. Justify your answer.

   (a) Social distancing gold-panning. Imagine a river network in which your team can pan for gold, but no two of you can stand in adjacent positions. You have some idea of the expected amount $w(v)$ of gold you will find at each location v, but must decide in which locations your team should look.

   Input: A rooted tree T, with root vertex $r \in V(T)$, and vertex weights $w\colon V(T) \to \mathbb{R}_{\geq 0}$

   Output: A subset of vertices $P \subseteq V(T)$ such that no two vertices in P are adjacent, and maximizing the value $\sum_{v \in P} w(v)$.

   Answer:

   In this problem, we have a tree, T, such that we can't select nodes in adjacent positions. In other words, we cannot pick two locations directly linked with a path. To solve this problem, we can split our main tree, T, into sub-trees, S, with root node, N. These sub-trees are our sub-problems because the properties of the sub-problem are the same as the original problem. Because of our location restriction, we would only be able to pick every other row. This is because if we selected node, N, we wouldn't be able to select $N's$ children since they are directly linked. Therefore, we would have to select N's grandchildren. Similarly if we selected N's children over N, we wouldn't be able to select N's grandchildren, but rather, N's great-grandchildren. Our goal is to find the largest value selection, so in the form of a recurrence equation, this can be represented as the following (Let W(N) represent that maximum value of the sub-tree rooted at node N):

   $$W(N) = max(w(N) + w(N_{grandchildren}), w(N_{children}) + w(N_{great-grandchildren}))$$

   Then, this can recursively be called until it goes through the whole tree, T. We can say that $W(N)$ are solutions to overlapping sub-problems as we can use the solutions to those to generate solutions for new sub-problems. Because we have overlapping sub-problem and the optimal substructure property holds, we can use the recurrence formula above to yield an optimal solution for this problem.

(b) Counting Knapsack. Here, we are considering the knapsack problem, but rather than returning the value of the optimum knapsack, or the optimum knapsack set, we are asking for the *number* of different optimum knapsacks (which all therefore have the same value).

Input: A list $L = [(w_1, v_1), \ldots, (w_n, v_n)]$, and a threshold weight W.

Output: The count of max-value knapsacks. A max-value knapsack is a subset $S \subseteq \{1, \ldots, n\}$ such that (1) $\sum_{i \in S} w_i \leq W$, and (2) the value $\sum_{i \in S} v_i$ is maximum among all subsets satisfying (1). The output should be *how many* different optimal solutions S there are.

For instance, if $L = [(1, 1), (1, 2), (2, 2), (2, 3)]$ and $W = 2$, then the output would be 2, because there are two optimal solutions: taking either the first two items or the very last item results in a valid knapsack of value 3. (Note that this is *not* just the total number of valid knapsacks; in this case there is a third set that fits within the weight threshold, namely the singleton $\{(2, 2)\}$, but that set does not have optimal value.)

Notes and References

- Worked with other students in office hours