Assignment 5

● Graded

Group

Jenny Yuan
Collins Munene Kariuki
View or edit group

Total Points

33.75 / 37 pts

Question 1

CS Snack

9.75 / 10 pts

– **0 pts** Correct

– **0.5 pts** Ambiguous description, or not enough detail. (The grader cannot follow very well)

– **1 pt** Need to start at index 0 and set to 0 all indexes from that until the cheapest price for the DP solution to be able to function. (like if we had the value of $7, and snacks cost 3 at min and has val 2, we still need to realize that val(3) + val(3) + val(1) yields 2+2+0, not throwing a value error for val(1)

– **1 pt** No discussion of how to initialize the excitement table

– **0 pts** Something odd but no points off. Look at comments

✔ – **0.25 pts** Clear initialization step, but need to explicitly state that the base initialization for the excitement table is 0s, and set to 0 all indexes from index 0 until at least the cheapest price. This is necessary for the DP solution to be able to function. (like if we had the value of $7, and snacks cost 3 at min and has val 2, we still need to realize that val(3) + val(3) + val(1) yields 2+2+0, not throwing a value error for val(1).

– **1 pt** Explicitly state initialization

– **0 pts** Okay so you are supposed to say things like how large the table is and how you fill it up, but your algorithm is so impressively formatted I am not taking points off as it is easy enough to infer those answers.

– **0.5 pts** The maximum element is necessarily the last element, because you would never write less than the max in a cell because you could just throw away the extra money and use the higher value from a previous cell

– **1 pt** The recursive solution only needs a one-dimensional table with d+1 size.

– **0.25 pts** Length d+1 (because its 0 to d)

Question 2

Words

12 / 12 pts

✔ **– 0 pts** Correct

– 0.5 pts Ambiguous description, poor formatting, or not enough detail (aka grader is confused)

– 1 pt Mention where the solution can be found

– 0 pts Say where the solution is found, etc. explicitly next time as the problem asks, but the table is clear so its aight by me (but might not be for a different grader)

– 0.25 pts size n+1

– 0.5 pts The base case (empty) needs to be set to true in initialization

– 1 pt Initialize the values. One on the end must be true and the rest false (which end depends on your implementation)

Question 3

Party

12 / 15 pts

– 0 pts Correct

✔ **– 1.5 pts** $O(nk^2)$. Looping through the grandchildren for each node means that for each k child you iterate through each of their k children, making it $n * k^2$.

1

✔ **– 1.5 pts** No way to keep track of if you already checked a parent.

2

– 1 pt Recursion should be called on both options regardless of which sum is greater.

– 1 pt Recursive function should be called on the grandchildren too

– 1.5 pts Should be adding parents of current node to queue if they are not already in the queue.

– 0.5 pts Should state that you get final answer in root node.

– 0 pts Click here to replace this description.

– 0.25 pts children(children(node))

– 2 pts Should be comparing whether or not to include current node and recursively calling function on children and grandchildren

Question assigned to the following page: 1

CS140 - Assignment 5

Due: Sunday, 2/25 at ~~10pm~~ 11:59pm

https://xkcd.com/2584/

For this assignment, you may (and are encouraged to) work with a partner.

For the dynamic programming solutions, in addition to the algorithm, make sure to explicitly state:

- What the table looks like (size and range of values).
- How you initialize the table, i.e., any starting values
- How you fill the table in, i.e., what indices you start at and how you proceed.
- Where the final answer is.

This can either be done with pseudocode or in plain language.

1. CS Snack **[10 points]**

   The CS liaisons have asked for your help with shopping for the CS snack. There are n possible snack options that you can choose from and you have d dollars to spend. Each snack option has been voted on with how excited the students are about the snack: $e_1, e_2, ..., e_n$, with higher being better. Each snack also has a price, $p_1, p_2, ..., p_n$. Your job is to pick which snacks so as to optimize the sum of the excitement of the snacks chosen while spending $\leq d$ dollars. Note that you can choose a given snack option multiple times.

   (a) **[8 points]** Give a dynamic programming solution that determines the maximum excitement sum, given the budget constraints (you don't need to explicitly state what snacks are chosen, just the overall sum of the excitement). Make sure to explicitly state the size of the table, which elements you fill in first, how you fill in the table, and where the solution is found.

   Answer: The essence of the DP approach is to consider our starting budget of d dollars and progressively determine the best snack (where "best" refers to the highest excitement level) we can afford after each purchase. For instance, after buying snack 1 for p_1 dollars, we evaluate which snack provides the maximum excitement with the remaining $d - p_1$ dollars. This process repeats with each subsequent purchase, reducing our budget each time by the cost of the chosen snack, until we either exhaust our funds or no snacks

Questions assigned to the following page: 1 and 2

are affordable with the remaining budget. This iterative decision-making process, where the optimal choice at each stage depends on the remaining budget, is characteristic of dynamic programming.

In the DP table, we initialize an array with indices ranging from 0 to d. We populate this table incrementally, at each index evaluating the most exciting snack purchase possible. The determination of which snack to buy at each price point is made by considering the excitement value of each snack plus the maximum excitement already recorded in the table for the remaining budget after that snack's cost. For example, to compute the excitement level at $d = 15$, we invoke our function 'excite(d)', which returns the highest excitement achievable with a 15-dollar budget. Specifically, 'excite(d)' computes the maximum of $e_1 + S(d - p_1), e_2 + S(d - p_2), ..., e_n + S(d - p_n)$, where e_i is the excitement value of snack i and $S(d - p_i)$ is the maximum excitement value retrievable from the DP table for the remaining budget after purchasing snack i.

Snacks that are too expensive for the remaining funds are excluded from the calculation.

Ultimately, the final answer—the maximum excitement that can be generated with the initial budget d—will be located at the last index of the DP array.

(b) **[2 points]** State the running time of your algorithm in terms of n the number of snack options and d the budget. $O(n * d)$.

2. Words **[12 points]**

You are given a string of characters $S = s_1, s_2, ..., s_n$ where all non-alphabetic characters have been removed (e.g. "thisisasentencewithoutanyspacesorpunctuation") and a function DICT(w, i, j), which takes as input a string w and two indices i and j and returns $true$ if the string $w_{i...j}$ is a dictionary word and $false$ otherwise.

(a) **[10 points]** Give a dynamic programming solution that determines whether the string S consists of a sequence of valid dictionary words. Make sure to explicitly state the size of the table, which elements you fill in first, how you fill in the table, and where the solution is found.

We initialize the table with n+1 being true. we create two for loops, one loop for i and one loop for j. For i loop, its a outer loop and we start the pointer from the last element and starts to move forward. For j loop, we set j = i and keep moving rightward to check if j is a word, j and j+1 a word ... till starts from j to the last element of the array is a word. Once we find a word, we check if j+1 is true, then we mark this as true. We keep moving j until we reach at the end of the string, if we didn't mark it as true, we mark this element at i as false. We continue doing this until i is at first element. Then we check the first element and if it is true then we return true else we return false. The size of the table is n+1, the n is the size of the input string, and you find the solution at the first index.

(b) **[2 points]** State the running time of your algorithm assuming calls to DICT are $O(1)$. the runtime is O(n^2).

3. Party **[15 Points]**

You've been asked to design an algorithm for deciding who to invite to a company party. The structure of the company can be described by a tree as follows: the CEO is at the root, below

Question assigned to the following page: 3

the root are VPs, below them are directors, below them are manages, etc., etc., until you get down to the leaves (summer interns). The tree is not necessarily binary; some non-leaf nodes may have one "child", others two, and others even more.

To make the party fun, we won't invite an employee along with their immediate supervisor (their parent in the tree). In addition, each person has been assigned a positive real number called their *coefficient of fun*. The goal is to invite employees so as to maximize the total sum of the coefficients of fun of all invited guests, while not inviting an employee with their immediate supervisor.

(a) **[4 points]** Describe a recursive algorithm for this problem (i.e. non-dynamic programming). Assume that the tree is represented as a collection of nodes with links from parents to children and also from children to parents. The tree is passed to you by giving you a reference to the root.

Answer: Let's define our function as 'maxFun', which starts by accepting a reference to the root node.

In the base case scenario, we encounter leaf nodes. For these nodes, the highest fun we can derive comes directly from their own fun value, as their inclusion does not affect any parent node. Thus, we retrieve and return this value, as opposed to a null fun value, in our pursuit to maximize total fun.

Moving on to the recursive step, each non-leaf node presents us with two scenarios: either we incorporate its fun value into our total, excluding its direct descendants, or we bypass it, considering only the fun values of its children. At each node, our aim is to capture the maximum cumulative fun obtainable by exploring both scenarios.

We establish two counters, let's name them 'option1' and 'option2'. 'option1' represents the total fun including the current node's value, initialized to the node's fun value. 'option2', initialized to zero, represents the total fun excluding the current node's value.

For 'option2', we iterate over the current node's children, summing up the 'maxFun' of each to 'option2'. This assumes we have a mechanism to list a node's children.

Conversely, 'option1' requires us to skip the immediate children due to the problem's constraint. Therefore, we iterate over each child, but instead of considering the children themselves, we delve into their children (the original node's grandchildren). Prior to this, a check is necessary to ensure we do not attempt to process leaves as parents. If a child is not a leaf, we recursively call 'maxFun' for each grandchild, summing the results to 'option1'.

Ultimately, we choose the greater of 'option1' or 'option2'—reflecting the maximum fun achievable—and return this value.

The function culminates by calling 'maxFun' on the root node, thereby setting in motion the recursive evaluation that will yield the maximum fun coefficient achievable for the entire tree.

(b) **[8 points]** Describe a DP algorithm for this problem. You may assume that each of the n nodes in the tree has a unique number between 1 and n associated with it. You may also assume that you have a function that will give you a list (array or linked list) of all of the leaves in the tree in time $O(n)$.

Question assigned to the following page: 3

Answer: We create an array named ‘maxFun‘ with a length equal to ‘n‘, where each element represents a node in the tree. Initially, we set all elements in ‘maxFun‘ to a default value, reflecting a minimal coefficient of fun of 1, to signify the starting point for each node’s maximum fun calculation.

Next, we populate a queue with the tree’s leaf nodes, obtained via a provided function, such as ‘leaves(tree)‘. This queue enables us to approach the dynamic programming (DP) problem from the bottom up, starting from the leaves and progressing towards the tree’s root. At each step, we evaluate, ”What is the highest coefficient of fun achievable up to this node?”

Upon dequeuing a node from the queue, we proceed to enqueue its parent(s), ensuring that we gradually move upwards through the tree. Concurrently, we evaluate the maximum fun coefficient for each node, adhering to the constraint that direct supervisors cannot be invited alongside their subordinates. Unlike the recursive approach that demands repetitive function calls, we consult the ‘maxFun‘ array for the pre-calculated fun coefficients of a node’s descendants. This direct lookup is an O(1) operation, significantly enhancing efficiency. Something also important to notice about this bottom-up approach is that since we start at the leaves, we automatically know what the maximum fun coefficient initially is, it’s just the fun value of the leaves

This method is notably more efficient than its recursive counterpart. It avoids redundant recalculations by leveraging previously computed values stored in the ‘maxFun‘ array. Starting with the leaves, where the maximum fun coefficient is directly given by their fun values, we build our way up to the root. The final answer—representing the maximum total fun achievable—is found at the root’s corresponding index in the ‘maxFun‘ array (the last index), culminating the DP process.

(c) **[3 points]** State the running time of your approach with respect to n the number of employees and k the maximum number of children any node has. **Answer**: $O(n * k)$