Homework 4 – COT5405

November 2021

1. **Peter is the owner of a Japanese sushi restaurant and today he invites you for dinner at his restaurant. In front of you are n sushi dishes that are arranged in a line. Each dish is different and they have different costs. Peter hopes that you can select the dishes you want, starting from the left to the right. However, there is a further restriction: when you select a dish, say A, the next dish you can select must cost higher than A. Design an $O(n^2)$-time algorithm to select the dishes so as to maximize the total costs.**

   This problem can be solved with DP/memoization using a 2D dp/memo matrix. In my solution I will use top-down DP (with memoization), but it can also be done iteratively using a DP matrix in a bottom-up fashion.

   Start by initializing a memo table to hold intermediate calculations. It will roughly be of size n^2; the first axis will represent whatever dish (current index, denoted `index`) we're currently deciding to select (or not), and the second axis will represent the dish we chose before this one (the previous index chosen, denoted `prev`).

   In this problem, recursion should end once we've looked at all n dishes, and we will start the recursion from the leftmost dish. In each call, we first check if our memo table has our current state (i.e. whether `memo[index][prev]` already has a value). If so, we can just return it without needing further computation. If it doesn't, we must compute and store it in the memo table. So, we go on to recursively compute the cost if we take the dish (first checking that it costs more than `prev` of course), and the max cost if we chose to skip the dish. Our memo table will then store the max of either taking or skipping the dish. We can keep track of the max as we recurse and return once it's finished.

   The time complexity of this approach is $O(n^2)$, which is the cost for filling the memo table. Since we don't do any recomputation (previously-seen states are just pulled from the memo table instead of recomputed), we incur $O(1)$ work for each unique state (and there are approx. n^2 of them).

   Below is some (unoptimized) pseudocode which details my approach.

Algorithm 1 Recursive memoization approach ($O(n^2)$ time with $O(n^2)$ space)

```
memo = [[null for i in [1, n]] for j in [1, n]]
maxcost ← −∞
procedure RECURSE(index, prev)
    if index ≥ len(arr) then
        return 0
    end if
    if memo[index][prev] ≠ ∅ then
        return memo[index][prev]
    end if
    take ← 0
    skip ← 0
    if arr[index] > arr[prev] then
        take ← arr[index] + RECURSE(index + 1, index)
    end if
    skip ← RECURSE(index + 1, prev)
    memo[index][prev] ← max(take, skip)
    return memo[index][prev]
end procedure
```

2. **There is a stair with n stages, and on each stage, there is a coupon to your favorite restaurant. Each coupon has an associated value which may be different. You can climb up 1, 2, or 3 stages in a step. Since you are in a hurry, you need to reach the top of the staircase in at most L steps, where $L \leq n$. When you visit a particular stage, you can collect the coupon on that stage. Note that L is a part of the input (means you cannot fix $L = n$ or so). Find a way to climb the stair within L steps so that you can collect coupons with the maximum total value. Your algorithm should run in $O(n^2)$ time.**

   Note: For this problem, I'm assuming coupon values are non-negative (i.e. taking a step can ONLY increase or maintain your "score"). This means that I will always try to take L steps. If negative coupons are allowed, then the solution would have to change, as it may be optimal to take less than L steps while still reaching the top (i.e. we may need to skip over some stairs with negative values).

   This problem can be solved with dynamic programming using a 2D dp matrix. The matrix will by $L \times n$, where L is the max. number of steps we can take and n is the total number of stairs. Our objective function (i.e. what each cell in the dp matrix represents) is simply the maximum cumulative coupon value we've gotten by our L^{th} step at the n^{th} stair. With this in mind, the main idea behind the algorithm is to build the matrix from the top left to the bottom right as we go along, and our answer will be somewhere in the final column of the matrix. If we take L steps, the final answer will be in the bottom right corner (otherwise, if we took less than L steps, we'd have to do a linear scan of the final column). This is because the final column represents our "score" by the time we've reached the top of the staircase (the n^{th} step), and the score we have by the time we've taken our L^{th} step should be the maximum possible.

   Initially, all matrix cells will be initialized to 0, after which we'll fill in our base cases. Since we can take 1, 2, or 3 steps, we can fill in the values for stairs 1, 2, and 3 before starting (these are our base cases). Thus, `dp[1][1]`, `dp[1][2]`, and `dp[1][3]` will be set to the coupon values of stairs 1, 2, and 3 respectively, since those are the stairs we can reach within 1 step. From there, we fill in the dp matrix row by row, by filling a cell with the max value of the previous 3 stairs in the previous step (i.e. the cells up 1 row and left 3 columns) plus the current stair's coupon value. See the example below for a walkthrough (assume the table continues past the bounds shown here, where the rows continue until an arbitrary L and the columns continue until an arbitrary n):

   Assume we have a staircase with $L = 3$ and the following coupon values (in order of stair number): $[3, 7, 9, 5, 1]$

3	7	9	0	0
0	0	0	0	0
0	0	0	0	0

Table 1: Base cases

3	7	9	0	0
0	10	16	14	0
0	0	0	0	0

Table 2: After first outer loop iteration

3	7	9	0	0
0	10	16	14	0
0	0	19	21	17

Table 3: Final iteration (we've reached taken our last possible step)

   As you may see, to fill in a particular cell, we only need the values of the cells up one row and up to three columns to the left, plus the value of the current stair's coupon. As we are trying to maximize the total coupon value, we take the max of those three cells and add it to the current stair's coupon value. This satisfies the optimal substructure property, allowing us to build up to the optimal final solution.

This final answer will thus be located in `dp[L][n]`, and we can confirm this is correct in our example (start from the ground (step "0"), go to stair 2 and collect 7, then go to step 3 and collect 9, then go to the final (n^{th}) step and collect 1 for a total of 17).

The final runtime for this algorithm is thus $O(3Ln)$, and since L is bounded by n, this is also bounded by $O(n^2)$.

3. **You have newly obtained the right to invest in a strange city called Trellisland. The shape of the city looks like a tree, such that each node in the tree has a building, and each edge in the tree is a road connecting two buildings. And in total, there are n buildings. You want to open stinky tofu stores in this city. According to the rules, each building can have at most one store, and each store cannot be adjacent to each other on the same road (otherwise, it becomes too stinky). You have done some preliminary study and can now tell the number of customers who will visit each building if a store is open there. Now, your target is to decide where to open the stores so as to maximize the total number of customers. Design an $O(n)$-time algorithm to achieve the above target.**

   This problem can also be solved using memoization/DP in a similar fashion to how we solved problem 1, just with different constraints/requirements. I'll be using the top-down memoization approach for convenience, but I won't be providing pseudocode again since it has the same structure (compute the answer for this state and store it in the memo table if not there already, otherwise pull the answer from the memo table and return).

   Start with the first node listed in the graph representation. In each recursive call, simulate both "taking" the current node (which represents putting a stinky tofu store in this building), and "not taking" the current node (no store in this building). If we take take the current node, recursively call again on the nodes after the neighboring nodes (to adhere to the store distance rule). If not, recursively call again on the neighboring nodes.

   When we return from a call, we'll know whether it was advantageous to take or skip a building (using a simple max), at which point we can mark it as taken or not taken in a separate array of nodes, which will be our final result.