CS 170 Homework 7

Due **10/17/2023, at 10:00 pm (grace period until 11:59pm)**

4-Part Solutions

For all (and only) dynamic programming problems in this class, we would like you to follow a 4-part solution format:

1. **Algorithm Description:** since dynamic programming algorithms can be difficult to explain, you should follow the template below to optimize clarity.
   (a) *Define your subproblem.* In words, define a function f so that the evaluation of f on a certain input gives the answer to the stated problem.

   You should clearly state how many parameters f has, what those parameters represent, what f evaluated on those parameters represents, and what inputs you should feed into f to get the answer to the stated problem.

   (b) *Provide your recurrence relation.* More precisely, give a recurrence relation showing how to compute f recursively, and make sure to provide base cases. If you need to use certain data structures to make computation of f faster, you should say so.

   (c) *Subproblem Ordering:* describe the order in which you should solve the subproblems to obtain the final answer.
2. **Proof of Correctness:** provide some inductive proof that shows why your DP algorithm computes the correct result.
3. **Runtime Analysis:** analyze the runtime of your algorithm.
4. **Space Analysis:** analyze the space/memory complexity of your algorithm.

1 Study Group

List the names and SIDs of the members in your study group. If you have no collaborators, you must explicitly write "none".

Solution: Got quite a lot of help during office hours from TAs about the egg drop and problem 5, but the rest I did on my own.

2 Non-Prefix Code

As we have learned in lecture, the Huffman code satisfies the *Prefix Property*, which states that the bit string representing each symbol is not a prefix of the bit string representing any other symbol. One nice property of such codes is that, given a bit string, there is at most one way to decode it back to a sequence of symbols. However, this is not true anymore once we are working with codes that do not satisfy the Prefix Property. For example, consider the code that maps A to 1, B to 01 and C to 101. A bit string 101 can be interpreted in two ways: as C or as AB.

Your task is to, given a bit string s, determine whether it is possible to interpret s as a sequence of symbols. The mapping from symbols to bit strings of the code will be given to you as a dictionary d (e.g., in the example, $d = \{A : 1, B : 01, C : 101\}$); you may assume that you can access each symbol in the dictionary in constant time.

(a) Describe an algorithm that solves this problem in at most $O(nk\ell)$ where n is the length of the input bit string s, k is the number of symbols, and ℓ is an upper bound on the length of the bit strings representing symbols. **Please provide a 4-part solution**.

Solution:

Algorithm Description: Let $S(i)$ indicate whether the first i character can be interpreted as a valid sequence of symbols.

For the recurrence relation, first let ℓ_k denote the length of the bitstring for the k-th letter. For every letter k, we need to look back ℓ_k letters, and determine whether the letter k fits within that string. Mathematically, this means:

$$S(i) = S(i - \ell_k) \wedge 1(\text{match } i - \ell_k \text{ to } i).$$

For that specific k. As for our base cases, $S(0)$ is set to true, and if $i - \ell_k < 0$, then $S(i - \ell_k)$ is set to false.

For the subproblem ordering, we want to start with $i = 0$, then increase the index i until we reach the length of s. If $S(s)$ is true, then return true. Otherwise, return false.

Proof of Correctness: We can prove this is correct via induction. Our base case can be a string of length 0, which the base case in our DP algorithm gives us true.

Now, suppose we know whether $S(i)$ can be interpreted as a valid sequence of symbols. We now prove that our algorithm gives the correct interpretation of $S(i+\ell_k)$. Given our recurrence relation, if $S(i)$ is false, then $S(i+\ell_k)$ will always be false, even if the next ℓ_k letters match. If $S(i)$ is true, then the truth value of $S(i + \ell_k)$ depends entirely on the next ℓ_k letters – if the letter matches, then we know that $S(i+\ell_k)$ can be validly interpreted, and our algorithm returns true to reflect that. Otherwise, we cannot use letter k, and our algorithm returns false as expected.

Another thing of note is that we don't need to keep track of exactly what the specific string is, but rather we only care about whether the string can be validly interpreted. Therefore, any $S(i + \ell_k)$ is sufficient.

Runtime analysis: At each subproblem, we are checking k letters, and each check takes at most $O(l)$ comparisons, so at each subproblme we have $O(kl)$ work. Given that the bitstring is of length n, then there are n subproblems, for a total runtime of $O(nkl)$.

Space Analysis: Really we only need to keep track $S(i)$ for every i from 0 to n, so we only need an array of length n. Thus, the space complexity is $O(n)$.

(b) **(Optional)** How can you modify the algorithm from part (a) to speed up the runtime to $O((n+k)\ell)$?

3 Ideal Targets

You are given a directed acyclic graph $G = (V, E)$ with unweighted edges. Every vertex $v \in V$ has an integer score $s[v]$. For a vertex v, we say that a vertex u is an ideal target for v if:

1. It is possible to go from v to u.
2. $s[u]$ is maximized.

In other words, out of all vertices that v can reach, u (i.e. v's ideal target) is the one with the maximum score.

Given the scores for all vertices, describe a linear-time algorithm to find the ideal targets for every vertex v. (Note that v can be its own ideal target.)

Just provide the algorithm description and runtime analysis. Proof of correctness and space complexity analysis are not required.

Solution:

Algorithm Description: Initially start with all $s[v] = \infty$. We start at the sink of the graph, and determine its optimal target. Since this is a sink, its optimal target will be itself (there are no out-edges in a sink). Then, for all nodes v_i connecting to the sink, find its ideal targets by comparing its value to that of the sink. If the sink's value is larger than that of any node, update it's value to be the value of the sink. Repeat this process by now considering all nodes connected to v_i and comparing their values to their corresponding v_i, update accordingly. Repeat until all nodes have a value.

Runtime Analysis: We can find the sink of the graph via DFS, which takes $O(|V| + |E|)$ time. Then, to assign ideal targets, we need to run the graph in reverse, so we reverse the edges, taking $O(|E|)$ time. Finally, we can assign ideal targets by running DFS one final time on the graph, taking $O(|V| + |E|)$ time. Therefore, in total:

$$O(|V| + |E|) + O(|E|) + O(|V| + |E|) = O(|V| + |E|)$$

4 Egg Drop

You are given m identical eggs and an n story building. You need to figure out the highest floor $\ell \in \{0, 1, 2, \ldots n\}$ that you can drop an egg from without breaking it. Each egg will never break when dropped from floor ℓ or lower, and always breaks if dropped from floor $\ell + 1$ or higher. ($\ell = 0$ means the egg always breaks). Once an egg breaks, you cannot use it any more. However, if an egg does not break, you can reuse it.

Let $f(n, m)$ be the minimum number of egg drops that are needed to find ℓ (regardless of the value of ℓ).

(a) Find $f(1, m)$, $f(0, m)$, $f(n, 1)$, and $f(n, 0)$. Briefly explain your answers.

Solution: I'll start with $f(0, m)$. There's only one floor, and since the floor ℓ must exist, then the egg must break at $\ell = 0$, so $f(0, m) = 0$.

For $f(1, m)$ we need to drop a single egg, from floor 1. If the egg breaks, then we know that $\ell = 0$. Otherwise, we know that $\ell = 1$.

For $f(n, 1)$, we only have a single egg, so the strategy would to start at floor 0, then go up the floors until the egg breaks, at which point we know that ℓ is one floor lower than the highest height that we dropped. This means that $f(n, 1) = n + 1$, at worst. We want the worst case for later on, so that's what we'll set this value to.

For $f(n, 0)$ this is impossible since we have no eggs, so we'd assign this value to ∞. (I could set this to anything, but setting it to ∞ makes the most sense in terms of the recursion.)

(b) Consider dropping an egg at floor x when there are n floors and m eggs left. Then, it either breaks, or doesn't break. In either scenario, determine the minimum remaining number of egg drops that are needed to find ℓ in terms of $f(\cdot, \cdot)$, n, m, and/or x.

Solution: If the egg doesn't break, then we know that there are only $n - x$ floors that the egg could possibly break on, and we have m eggs. Therefore, this number would be $f(n - x, m)$.

If the egg breaks, then we know that the critical floor is somewhere below us, so that means that we need to check the floors from 0 to $x - 1$. We've also lost an egg, so here we have $f(x - 1, m - 1)$.

(c) Find a recurrence relation for $f(n, m)$.

Hint: whenever you drop an egg, call whichever of the egg breaking/not breaking leads to more drops the "worst-case event". Since we need to find ℓ regardless of its value, you should assume the worst-case event always happens.

Solution: We follow the hint, and consider what happens when we drop an egg from every floor. For every floor, we check what would happen if the egg breaks or doesn't break, and choose the maximum of the two (since we're trying to find the "worst case event"). Repeat this for every floor, then find the minimum f across among all the floors. Mathematically:

$$f(n, m) = \min\{\max_{\text{all floors } x} \{f(n - x, m), f(x - 1, m - 1)\}\}$$

(d) If we want to use dynamic programming to compute $f(n, m)$ given n and m, in what order do we solve the subproblems?

Solution: We see that in our recurrence relation, at each floor x, each f is determined by either $f(n-x, m)$ or $f(x-1, m-1)$. Therefore, each subproblem depends on the previous subproblems by the number of eggs by m or $m-1$, and for the former case we need to check the floors above x and in the latter we check the floors below x. All this is to say that in order to solve the subproblems, we first iterate through the number of eggs, then within each iteration we iterate through all the floors of the building from the bottom to the top.

This is because when we call $f(n-x, m-1)$, we are checking the floors above, which are already checked if we're on iteration m (based on the proposed order). If we're calling $f(x-1, m)$, we're on the current iteration of m, and we need to know the f value for the floor m, so we need to go bottom up in this case.

(e) Based on your responses to previous parts, analyze the runtime complexity of your DP algorithm.

Solution: During each subproblem, we query n different values of f, and choose the minimum of these, so each subproblem takes $O(n)$ time. Then, for each m, there are n, subproblems, for a total of $O(n^2 m)$ time.

(f) Analyze the space complexity of your DP algorithm.

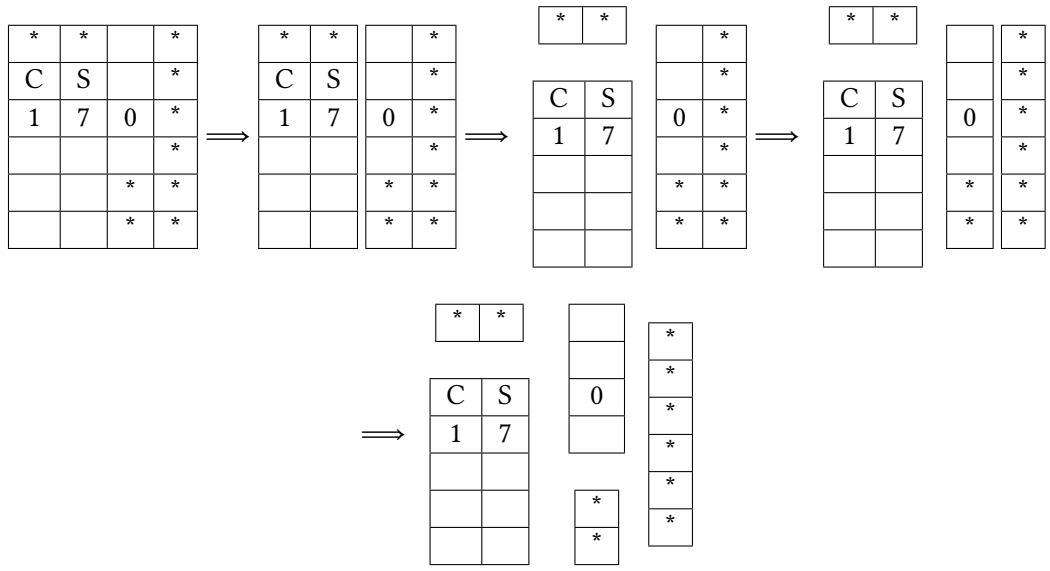
Solution: For each floor we have m values (one for each quantity of egg), so there are $O(nm)$ values in total to keep track of.

(g) (**Extra Credit**) Is it possible to modify your algorithm above to use less space? If so, describe your modification and re-analyze the space complexity. If not, briefly justify.

5 My Dog Ate My Homework

One morning, you wake up to realize that your dog ate some of your CS 170 homework paper, which is an $m \times n$ rectangular grid of squares. Some of the squares have holes chewed through them, and you cannot use paper that has a hole in it. You would like to cut the paper into pieces so as to separate all the tattered squares from all the clean, un-bitten squares. You want to do this so that you can save as much as your work as possible.

For example, shown below is a 6×4 piece of paper where the bitten squares are marked with *. As shown in the picture, one can separate the bitten parts out in exactly four cuts.



(Each *cut* is either horizontal or vertical, and of one piece of paper at a time.)

Design a DP based algorithm to find the smallest number of cuts needed to separate all the bitten parts out. Formally, the problem is as follows:

> **Input:** Dimensions of the paper $m \times n$ and an array $A[i, j]$ such that $A[i, j] = 1$ if and only if the ij^{th} square has holes bitten into it.
>
> **Goal:** Find the minimum number of cuts needed so that the $A[i, j]$ values of each piece are either all 0 or all 1.

(a) Define your subproblem.

Hint: try making any arbitrary cut. What two subproblems do you now have?

Solution: Let $S(x_1, y_1, x_2, y_2)$ denote the minimum number of cuts needed to separate the bitten parts out for a rectangle spanning from $[x_1, x_2]$ and $[y_1, y_2]$. By construction, the point (x_1, y_1) is the top left of the rectangle, and (x_2, y_2) is the bottom right. Here, the point $(x, y) = (0, 0)$ is the top left corner of the rectangle, and $(x, y) = (m, n)$ is the bottom right.

(b) Write down the recurrence relation for your subproblems. A fully correct recurrence relation will always have the base cases specified.

Solution: We'll start with the base cases. If the rectangle has no bitten parts (so no 1's), then the

number of cuts is 0. Similarly, if the rectangle has only bitten parts (so only 1's), then the number of cuts is also 0. There are no other base cases.

In terms of the recurrence relation, in order to find $S(x_1, y_1, x_2, y_2)$, we make an arbitrary cut, which now turns our single rectangle into two smaller ones. Suppose we cut along x_i, then the subproblmes would be $S(x_1, y_1, x_i, y_2)$ and $S(x_i, y_1, x_2, y_2)$. For the remainder of my solution though, we'll call these $S(L)$ and $S(R)$. Then, the number of cuts required, provided we make this arbitrary cut will be $S(L) + S(R) + 1$, where we add 1 to denote this arbitrary cut. So to find the minimum number of cuts, we need to iterate through all possible cuts, and compute this $S(L) + S(R) + 1$ for every cut. Mathematically:

$$S(x_1, y_1, x_2, y_2) = \min_{\text{all cuts}} \{S(L) + S(R) + 1\}$$

(c) Describe the order in which we should solve the subproblems in your DP algorithm.

Solution: To solve each subproblem, notice that they depend on all the possible ways we can make a cut. One way to go about solving the subproblems is to solve the problem by solving the rows individiually, forming "strips." Once each "strip" is complete, we can now start joining these strips together, by joining the 1st row with the 2nd, then join the combination of this with the third, and so on, until we've joined everything.

This is not the only order we could have gone, doing this column by column is also ok.

(d) What is the runtime complexity of your DP algorithm? Provide a justification.

Solution: During each subproblem, it checks through all the ways to make a valid cut, and for a rectangle of dimension mn there are mn valid cuts. Therefore, there is $O(mn)$ work at every step. The number of subproblems is the number of points (x_1, y_1) and (x_2, y_2) such that $x_1 < x_2$ and $y_1 < y_2$. There are $\binom{m}{2} \in O(m^2)$ of x coordinates, and $\binom{n}{2} \in O(n^2)$ y-coordinates that satisfy this, so the total runtime will be:

$$O(mn)O(n^2)O(n^2) = O(m^3n^3)$$

(e) What is the space complexity of your algorithm? Provide a justification.

Solution: Here, the most amount of space we'll ever need is when counting the 1x1 squares, which has $O(mn)$ space complexity. Once we start combining subproblems, we no longer need these 1x1 squares, so the space used reduces over time. Therefore, the overall space complexity is $O(mn)$.

6 Coding Questions

For this week's coding questions, we'll implement dynamic programming algorithms to solve two classic problems: **Longest Increasing Subsequence** and **Edit Distance**. There are two ways that you can access the notebook and complete the problems:

1. **On Local Machine**: `git clone` (or if you already cloned it, `git pull`) from the coding homework repo,

   `https://github.com/Berkeley-CS170/cs170-fa23-coding`

   and navigate to the `hw07` folder. Refer to the `README.md` for local setup instructions.

2. **On Datahub**: Click here and navigate to the `hw07` folder if you prefer to complete this question on Berkeley DataHub.

Notes:

- *Submission Instructions:* Please download your completed submission `.zip` file and submit it to the Gradescope assignment titled "Homework 7 Coding Portion".
- *OH/HWP Instructions:* Designated coding course staff will provide conceptual and debugging help during office hours and homework parties.
- *Edstem Instructions:* Conceptual questions are always welcome on the public thread. If you need debugging help first try asking on the public threads. To ensure others can help you, make sure to:

  1. Describe the steps you've taken to debug the issue prior to posting on Ed.
  2. Describe the specific error you're running into.
  3. Include a few small test cases, alongside both the output you expected to receive and your function's actual output.

  If staff tells you to make a private Ed post, make sure to include *all of the above items* plus your full function implementation. If you don't provide them, we will ask you to provide them.

- *Academic Honesty Guideline:* We realize that code for some of the algorithms we ask you to implement may be readily available online, but we strongly encourage you to not directly copy code from these sources. Instead, try to refer to the resources mentioned in the notebook and come up with code yourself. That being said, we **do acknowledge** that there may not be many different ways to code up particular algorithms and that your solution may be similar to other solutions available online.