Problem Set 8

All parts are due Friday, November 15 at 6PM.

Name: Diego Escobedo

Collaborators: Noah Lee, Ritaank Tiwari, Ellery Rajagopal

Problem 8-1. Assuming we represent the pickled pepper garden as an array A containing p_i at position i:

Subproblems: the subproblem DP(i) is going to be the minimum over the sum of p_i's picked in $A[:i]$

Recurrence: Since we have two options, namely whether to pick pepper i or not to pick it, we want to consider the two possible outcomes. In the case that we pick pepper i, the cost is $A[i]$ + DP(i-1). Further, if we don't pick it and have to pick the previous and the next, then the cost is $A[i-1]$ + DP(i-2) (we don't need to do DP(i-1) because we already picked it in this scenario, and we don't include $A[i+1]$ because we are imagining that A only goes up to i). Additionally, we also need to make sure we don't double count A[i-1] so we need to be keeping track of which pickles we have already picked. This relation is acyclic because our decision to pick or not to pick a given patch is decided exclusively by our decisions later down the line, meaning if we represented our decisions as a tree then there would be no back edges. Therefore, we have no cycles.

Base Case: DP(x) = 0 if x is less than 0, because it means we are not longer considering the smallest building blocks of iput and have

Solution: DP(len(A) - 1)

Running time: We have n subproblems, and in each one we are doing a constant amount of work. Therefore, the whole algorithm runs in O(n) time.

Problem 8-2. Assuming we represent the list of items as parallel arrays V (for value) and C (for cost):

Subproblems: the subproblem DP(i, k) is going to be the maximum value of items in $V[:i]$ while keeping the cost of the same items in $C[:i]$ under k.

Recurrence: Since we have two options, namely whether to include item i in the objects we want to create out of melted gold, we want to consider the two possible outcomes. In the case that we include item i, the reward is is $V[i]$ + DP(i-1, k - $C[i]$). Note that this is only valid if $k \geq C[i]$. Further, if we don't include it, then the reward is DP(i-1, k). This relation is acyclic because our decision to pick or not to pick an item is decided exclusively by our decisions later down the line, meaning if we represented our decisions as a tree then there would be no back edges. Therefore, we have no cycles.

Base Case: The base case is DP(x) when x is less than 0. In that case, DP(x) would be 0.

Solution: DP(len(V) - 1,n)

Running time: We have n subproblems, and in each one we are doing a constant amount of work. Therefore, the whole algorithm runs in O(n) time.

Problem 8-3. We start off the problem by reorganizing the given blocks. Because there are three of every block, and because we couldn't stack two blocks together on top of each other if they were in the same orientation anyway, then we can change the orientations of these blocks to make sure that we consider all possible orderings of them. So, if we are given a list of blocks, and each block has 3 copies represented by the tuple (w,l,h), then we would represent the three copies with (w,l,h),(w,h,l),(h,l,w). Now, we can create a new list of these modified tuples, sort it, and simply do a modified longest increasing subsequence algorithm weighting by the height of the block. At this point, it is now a DP problem.

An important thing we have to consider is how we're going to sort this list of tuples. The simplest is to sort by the surface area of the block, or in other words, in an array A of block tuples, sorting by A[i][0]*A[i][1]. This makes sense, because if we sort it by surface area, then we know for a fact that a block that has a higher surface area cannot be stacked on top of one with a lower one. However, if a block has less surface area than an other, it is not guaranteed to fit, so we check for that condition in our LIS code. NOTE THAT WE ARE SORTING FROM BIGGEST TO SMALLEST.

Subproblems: the subproblem DP(i) is going to be the maximum height of the optimal block configuration in $A[: i]$.

Recurrence: To solve DP(i), if we have a tuple (w_i, l_i, h_i) at position i, then the second to last tuple in a subsequence of blocks must be (w_j, l_j, h_j) for $j < i$ satisfying ($w_j > w_i$ and $l_j > l_i$) or ($l_j > w_i$ and $w_j > l_i$). So:

DP(i) = max($DP(j) + A[j][2]$ for all j where $0 \leq j < i$ and $w_j > w_i$ and $l_j > l_i$) or ($l_j > w_i$ and $w_j > l_i$)). Note that $A[j][2]$ is the same as h_j.

Base Case: The base case is DP(x) when x is less than 0. In that case, DP(x) would be 0.

Solution: DP(len[A] - 1)

Running time: The sorting can be done in nlogn if we do merge sort. The modified LIS has n subprolems, and each one is computed in n time, so it is overall $O(n^2)$.

Problem 8-4. The first thing we have to do is to store all the lines in B in a hash table, keyed by the actusl content of the line (which is bounded by k) and valued by the index of the line in B. This will allow us to determine if the given line is in a certain position in constant time:

Subproblems: the subproblem DP(i,j) is going to be the minimum number of edits expended in turning $A[:i]$ into $B[:j]$.

Recurrence: We have a great feature in that if $A[i]$ is equal to $B[j]$, then we are done with that line and we can move on. If they are not equal, then we can thnk of the edits as things we're trying to do to them to turn them equal to each other. The 4 operations we could do would be to insert (which adds $B[j]$ to the end of A and then deletes $A[i]$ and $B[j]$, cost of 1), delete (removes $A[i]$, cost of 1), replace (which uses insert and delete and tehrefore has a cost of 2), and swap (which is free, and changes the order of $A[i-1]$ and $A[i]$, though this only works if i is greater than or equal to 1. So our recurrence looks like:

if $A[i]$ is equal to $B[j]$: DP(i,j) = DP(i-1, j-1)
else: DP(i,j) = min(DP(i-1,j) + 1, DP(i,j-1) + 1, DP(i-1,j-1) + 2, DP(i-2,j-2)), which respectively correspond to deletion, insertion, replacement, and swapping.

Base Case: DP(*,0) = 0 for all * (this means we're done), and DP(0,j) = j (it means this many more edits have to be done because we're done editing all we can in A.

Solution: DP(len(A), len(B))

Running time: We have n^2 subproblems, and in each one we are doing a constant amount of work. However, our hashtable was created in $O(kn)$. Therefore, the whole algorithm runs in $O(kn + n^2)time$.

Problem 8-5.

(a) Subproblems: In this case, we actually have two subproblems. Subproblem X(i,j) is the maximum amount of mushrooms that can be picked from the start to i,j. Subproblem Y(i,j) is the of optimal quick paths from the start to i,j

Recurrence: First, lets recall that because we only have 2n-1 moves, if we ever move left or up, we will not be able to reach the objective. Therefore, we can only move down or right. X(i,j) has three cases:

If $A[i][j]$ is a tree: 0
If $A[i][j]$ is empty: max($X(i-1,j), X(i,j-1)$)
If $A[i][j]$ is a mushroom: max($X(i-1,j)+1, X(i,j-1)+1$)

Y(i,j) has four cases:
If X(i-1,j) = X(i,j) - 1 and there is a mushroom on i,j: sum(Y(i-1,j)) If X(i-1,j) = X(i,j) and there is not mushroom on i,j: sum(Y(i-1,j)) If X(i,j-1) = X(i,j) - 1 and there is a mushroom on i,j: sum(Y(i,j-1)) If X(i,j-1) = X(i,j) and there is not mushroom on i,j: sum(Y(i,j-1))
The first two cases means that there is an optimal path from i-1,j to i,j, which means we would like to go back to that space to explore whiter we could reconstruct an optimal path from there.
Base Cases: X(i,j) and Y(i,j) are 0 if i and j are not in $[0, n-1]$. X(0,0) is 0 because we are assuming there are no mushrooms at the start. Y(0,0) is 1 because we have found a succesful path with max mushrooms. Y(i,j) is 0 if there is a tree at i,j.

Solution: Y(n,n)

Running time: There are $O(n^2)$ subproblems, and each one has constant work, so we can do this is $O(n^2)$ time.

(b) Submit your implementation to `alg.mit.edu`.