CSE 40113: Algorithms
Homework 3

You may complete this homework in groups of 3 or less students. Note that the integrity policy applies: your group should write up your own work, although you're welcome to work on the problems in a larger group. If you have any questions, please re-read both the homework guidelines and the academic integrity policy carefully, and then come discuss any questions or concerns with me.

Required Problems

1. You are a professional robber planning to rob houses along a street. Each house has a certain amount of money stashed, which you are able to make an accurate estimate of based on publicly available information on the internet. You need to decide which houses to rob, where the only constraint stopping you from robbing each of them is that adjacent houses have security systems connected and it will automatically contact the police if any three adjacent houses were broken into on the same night. (You may assume all houses have positive values.)

   (a) Give a (small) example of why choosing the largest value house first (in other words, being greedy) will not necessarily yield the most money.

   Solution: Consider 3 houses, where house 1 is worth 3, house 2 is worth 4, and house 3 is worth 3. A greedy choice of the largest would give a total value of 4, but if you rob houses 1 and 3 instead, you could get a total value of 6.

   ■

   (b) Write a recursive formula or expression for the maximum amount of money you can gain.

   Solution: Suppose the estimated value of each house is stored in an array $H[1..n]$. Let $\text{Opt}(i)$ = the maximum amount of profit I can get for houses i through n. Our backtracking recursion will be to consider house i, which could be included or skipped. If you skip, you can just lookup $\text{Opt}(i+1)$, since there is no constraint to skip anything. If you take house i, your profit will increase by $H[i]$, but you must skip either house $i+1$ or house $i+2$. You can encode these possibilities recursively in several different ways, all of which receive full credit - here is one example:

$$\text{Opt}(i) = \begin{cases} 0 & \text{if } i > n \\ \max(\text{Opt}(i+1), H[i] + H[i+1] + \text{Opt}(i+3), H[i] + \text{Opt}(i+2)) & \text{otherwise} \end{cases}$$

   Of course, other recursive formulations of this problem are possible, and worth full credit - for example, you could let $\text{Opt}(i)$ be the maximum amount in houses 1 through i, with $i = 0$ as the base case. Or, you could keep two arrays and store two values per house, for with and without, and lookup appropriate values to ensure that if i is stolen (the "with" value), only one of $i+1$ and $i+2$ can also be stolen.

   ■

(c) Given an array Houses$[1 \ldots n]$, where Houses$[i]$ is the amount of money estimated in the i^{th} house, design an algorithm to calculate the maximum amount of money you can rob tonight without alerting the police.

Solution: We can turn our recursive formulation into a dynamic program by noting that for each house, we need to store a single value Opt(i). We can fill from n down to 1, where each value i needs to look at the three later values, perform the appropriate additions, and take a max. We get pseudocode as follows:

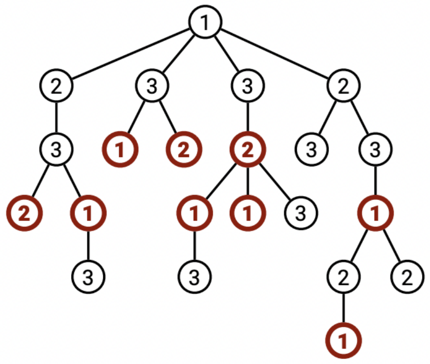
```
Rob($H[1..n]$)
  Initialize an array Opt$[1..n]$ with all 0's
  Opt$[n] \leftarrow H[n]$
  Opt$[n-1] \leftarrow H[n-1] + H[n]$
  Opt$[n-2] \leftarrow \max \text{Opt}[n-1], H[n-2] + H[n-1], H[n-2] + H[n]$
  for $i \leftarrow n-3$ down to 1
      Opt$[i] = \max$
  return Opt$[1]$
```

■

2. Since so few people came to last year's holiday party, the president of Giggle decides to give each employee a present instead this year. Specifically, each employee must receive on the three gifts: (1) an all-expenses-paid sixweek vacation anywhere in the world, (2) an all-the-pancakes-you-can-sort breakfast for two at Jumping Jack Flash's Flapjack Stack Shack, or (3) a burning paper bag full of dog poop. However, corporate regulations prohibit any employee from receiving exactly the same gift as his/her direct supervisor. Any employee who receives a better gift than his/her direct supervisor will almost certainly be fired in a fit of jealousy. As Giggle's social party czar, it's your job to decide which gift each employee receives.

   More formally, you are given a rooted tree T, representing the company hierarchy, stored as a data structure where you have saved the root as $T.root$. For each node v in the tree, there is a list of children saved in an array. You want to label the nodes of T with integers 1, 2, or 3, so that every node has a different label from its parent. The cost of an labeling is the number of nodes with smaller labels than their parents.

   See below for an example of such a tree, which has cost 9, where all fired employees are shown in bold (since they have values lower than their direct supervisor.)



   (a) Give a recursive formulation to calculate the minimum cost labeling rooted at a node v. (Hint: take inspiration from section 3.10 here.)

Solution: At each node, we can consider all possible labelings: 1, 2, or 3. For each, we will have different constraints on the children, as they must be one of the two other numbers, and some number can/will be fired as a result depending on number. So one recursive formulation is the following:

For node v and label i, where $i \in \{1, 2, 3\}$, let Holiday(v, i) = the minimum number of people that will be fired in the optimal labeling of the subtree rooted at v if v is given label i. If v is a leaf, then Holiday$(v, i) = 0$ for all $i = 1, 2$, and 3.

Otherwise,

$$\begin{aligned}
\text{Holiday}(v,1) &= \sum_{w \text{ a child of } v} \min\{\text{Holiday}(w,2), \text{Holiday}(w,3)\} \\
\text{Holiday}(v,2) &= \sum_{w \text{ a child of } v} \min\{(1+\text{Holiday}(w,1)), \text{Holiday}(w,3)\} \\
\text{Holiday}(v,3) &= \sum_{w \text{ a child of } v} \min\{(1+\text{Holiday}(w,1)), (1+\text{Holiday}(w,2))\}
\end{aligned}$$

Finally, the min cost labeling for node v is simply the minimum of Holiday$(v, 1)$, Holiday$(v, 2)$, and Holiday$(v, 3)$.

Correctness of this formula: Proof by induction on size of tree.

Base case: If v is a leaf, we correctly return 0 for all values $i = 1, 2, 3$.

Inductive Hypothesis: Assume our formulation for Holiday$(v, 1)$, Holiday$(v, 2)$, and Holiday$(v, 3)$. is correct for trees with $< n$ vertices.

Inductive Step: Consider a tree with n vertices.

■

(b) Design and analyze an algorithm to compute the minimum-cost labeling of T, so that the fewest number of people are fired. (Yes, you may send the president a flaming bag of dog poop.)

Solution: Our recursive formulation will be built from our recurrence (so yours may look different if you had a slightly different recursive formulation!). Here, I'll chose to store 3 values at every node v: v.label1 will store the minimum number fired assuming v gets label 1, v.label2 the minimum if v gets label 2, and v.label3 the minimum if labeled 3. We can fill these in at a leaf easily based on our recursive formulation, and for every node v, we can calculate the 3 values needed assuming all child values are already filled in in time proportional to the number of children, since we do a total of 6 lookups in order to fill in our 3 values in the recursion above.

We will calculate the values in a postorder traversal, so that children are calculated before the parent:

```
Holiday($v$)
  v.label1 $\leftarrow$ 0
  v.label2 $\leftarrow$ 0
  v.label3 $\leftarrow$ 0
  for each child $w$ of $v$
      v.label1 += min{ w.label2, w.label3 }
      v.label2 += min{1+ w.label1, w.label3 }
      v.label3 += min{1+ w.label1, 1+w.label2 }
  return min{ v.label1, v.label2, v.label3 }
```

Our "main" function would then just call Holiday(root).

Runtime: This is a postorder traversal, which takes $O(n)$ time total, as the time spent at each node is proportional to the number of children. If we have n nodes total, this will take a total of $O(n)$ time; even though the time spent at a given node is not constant, each node is only the child of one parent, so in total we still get $O(n)$ lookups across the entire algorithm.

■

3. Recall that a *subsequence* of an array is a subset of the elements in the array in the same order. (So: a subset in the same order, but they don't have to be next to each other in the array.) For example, 1, 1 5 3, and 2 1 9 5 8 7 2 6 5 3 are all subsequences of $A = [2\ 1\ 9\ 5\ 8\ 7\ 2\ 6\ 5\ 3\ ]$, where each $A[i]$ is a single digit.

   A sequence $X[1..n]$ is called *oscillating* if $X[i] < X[i+1]$ for all even i, and $X[i] > X[i+1]$ for all odd i. Describe an efficient algorithm to compute the length of the longest oscillating subsequence of an arbitrary array A of integers.

Solution: There were two ways to approach this, both akin to the Longest Increasing Subsequence problem. Since we want an oscillating sequence, we don't just need to remember the previous element chosen, but also need to know if the next element should be larger or smaller than the previous. So, one recursive formulation is to let LOS-up(i, j) be the longest oscillating sequence in $A[j..n]$, where $A[i]$ was the last element in the subsequence and the next value should increase, and LOS-down(i, j) be the longest oscillating sequence in $A[j..n]$, where $A[i]$ was the last element in the subsequence and the next value should decrease. Then, we can choose to match (in which case you switch to the other version of LOS, since you want to go the opposite way in the next term), or we don't, and choose the best of the two:

$$\text{LOS-up}(i,j) = \begin{cases} 0 & \text{if } j > n \\ \text{LOS-up}(i,j+1) & \text{if } A[i] \geq A[j] \\ \max(\text{LOS-up}(i,j+1), 1 + \text{LOS-down}(j,j+1)) & \text{otherwise} \end{cases}$$

$$\text{LOS-down}(i,j) = \begin{cases} 0 & \text{if } j > n \\ \text{LOS-down}(i,j+1) & \text{if } A[i] \leq A[j] \\ \max(\text{LOS-down}(i,j+1), 1 + \text{LOS-up}(j,j+1)) & \text{otherwise} \end{cases}$$

This means we will need to store our entries in two $O(n^2)$ tables:

```
LongestOscillatingSubsequence($A[1..n]$):
  $A[0] \leftarrow -\infty$
  for $i \leftarrow 1$ to $n$LOS-up$[i, n+1] \leftarrow 0$
      LOS-down$[i, n+1] \leftarrow 0$
  for $j \leftarrow n$ down to 1
      for $i \leftarrow 0$ to $j$
          if $A[i] \geq A[j]$
              LOS-up$[i, j] \leftarrow$ LOS-up$[i, j+1]$
          else
              LOS-up$[i, j] \leftarrow \max($LOS-up$[i, j+1], 1+$ LOS-down$[i, j+1])$
          if $A[i] \leq A[j]$
              LOS-down$[i, j] \leftarrow$ LOS-down$[i, j+1]$
          else
              LOS-down$[i, j] \leftarrow \max($LOS-down$[i, j+1], 1+$ LOS-up$[i, j+1])$
  return LOS-up$[0, 1]$
```

Correctness of the recursion (and algorithm): induction on j

Base case: we correctly calculate 0 if there is no sequence left ($j > n$), no matter the value of i and no matter if we're in LOS-up or LOS-down.

Inductive hypothesis: Assume the algorithm works correctly for LOS-up$(i, j + 1)$ and LOS-down$(i, j + 1)$, calculating correct the longest increasing subsequence where $A[i]$ is the last element included, and we need the next element to go either up or down (respectively). Now consider LOS-up(i, j) and LOS-down(i, j) for any i. In either recurrence, we try both including the element at $A[j]$ and not including it, assuming $A[j]$ is appropriately larger or smaller than the previously chosen element (based on which recurrence it is in). If we include, then the recursion needs to switch to the other recurrence, as the next element oscillates in the other direction. Since we try all possibilities with regards to $A[j]$ and the inductive hypothesis says that we can compute the correct answer for $j+1$ calls, we will generate the correct value.

In the end, we return LOS-up$[0, 1]$, as that stores the length of the longest oscillating sequence in $A[1..n]$, so we ensure the first element chosen for the subset is larger than the second.

Runtime and Space: We have two $n \times n$ tables, so our space is $O(n^2)$. The nested loop takes $O(n^2)$ time, since each iteration fills in a cell entry of each table in $O(1)$ time.

■

4. Sample Solved Problem: A shuffle of two strings X and Y is formed by interspersing the characters into a new string, keeping the characters of X and Y in the same order. For example, the string BANANAANANAS is a shuffle of the strings BANANA and ANANAS in several different ways:
BANANAANANAS, BANANAANANAS, or BANANANANAS.

   Similarly, the strings PRODGYRNAMAMMIINCG and DYPRONGARMAMMICING are both shuffles of DYNAMIC and PROGRAMMING:
PRODGYRNAM AMMIINCG and DYPRONGARMAMMICING.

   Given three strings $A[1..m]$, $B[1..n]$, and $C[1..m+n]$, describe and analyze an algorithm to determine whether C is a shuffle of A and B.

   Solution:

Recursive formulation: We define a boolean function $Shuf(i, j)$, which is True if and only if the prefix $C[1..i+j]$ is a shuffle of the prefixes $A[1..i]$ and $B[1..j]$. This function satisfies the following recurrence:

- Shuf(i, j) = true if $i = j = 0$
- Shuf$(0, j-1)$ AND $(B[j] = C[j])$ if $i = 0$ and $j > 0$
- Shuf$(i-1, 0)$ AND $(A[i] = C[i])$ if $i > 0$ and $j = 0$
- (Shuf$(i-1, j)$ AND $(A[i] = C[i+j])$) OR (Shuf$(i, j-1)$ AND $(B[j] = C[i+j])$) if $i > 0$ and $j > 0$

The proof that this formulation is correct can be shown via induction: if you're considering the $i + j^{th}$ character of C, it must be from either $A[i]$ or $B[j]$. We are trying both options, and returning true if either works. The base cases handle either A or B being empty, in which case either we've matched everything (and both are 0) or we must exactly match the rest of C to which ever string is left.

Dynamic programming: We need to compute Shuf(m, n); if it is true, then we can shuffle the entire strings A and B into a string C. Since Shuf[i,j] needs to look up values Shuf[i-1,j] and Shuf[i,j-1] (as well as comparing some values from the 3 arrays), we can memoize all values into a two-dimensional array Shuf[0..m][0..n]. Each array entry Shuf[i, j] depends only on the entries immediately below and immediately to the right: Shuf[i-1, j] and Shuf[i, j-1]. Thus, we can fill the array in standard row-major order.

```
SHUFFLE?($A[1..m]$, $B[1..n]$, $C[1..m+n]$):
  $Shuf[0,0] \leftarrow$ TRUE
  for $j \leftarrow 1$ to $n$
      $Shuf[0,j] \leftarrow Shuf[0,j-1] \wedge (B[j] = C[j])$
  for $i \leftarrow 1$ to $n$
      $Shuf[i,0] \leftarrow Shuf[i-1,0] \wedge (A[i] = B[i])$
      for $j \leftarrow 1$ to $n$
          $Shuf[i,j] \leftarrow$ FALSE
          if $A[i] = C[i+j]$
              $Shuf[i,j] \leftarrow Shuf[i,j] \vee Shuf[i-1,j]$
          if $B[i] = C[i+j]$
              $Shuf[i,j] \leftarrow Shuf[i,j] \vee Shuf[i,j-1]$
  return $Shuf[m,n]$
```

The algorithm runs in $O(mn)$ time, and (if we keep the entire 2d arrays) takes the same amount of space.

Note that this can be improved to $O(m)$ (or $O(n)$) by keeping only two rows: the one we are currently filling in, and the row immediately preceding it.