This homework is due by 11:59PM on Thursday, Mar 29th. You must submit the homework via the course page on T-Square.

Homework Policies:

- Your work will be graded on correctness and clarity. Write complete and precise answers.
- You may collaborate with up to *three* other classmates on this problem set. However, you must *write your own solutions* in your own words and *list your collaborators.*
- You may portions of our two textbooks that we have covered. The purpose of these homeworks is NOT only to teach you the algorithms that we cover, but also to teach you how to problem-solve. Therefore using *ANY* outside resources like the internet, previous students etc. is *not* allowed!
- **Optional** questions are for you to practice and learn the basics before you advance. They may ask you to follow the steps of an algorithm covered in class, or review material from an earlier class. Solutions to these problems will not be graded, but you must be are able to do them, and you are responsible for material that is addressed by these optional questions.
- **Basic** questions *must be solved and written up alone*; you may not collaborate with others. These will help you build the framework needed to solve the harder questions. Please do these first, before meeting in groups, for both you and your group's benefit.
- **Group** questions may be solved in collaboration with up to ***three*** other classmates. However, you must still *write up your own solutions* alone and in your own words. These will form the bulk of your homework questions.
- **Bonus** questions are like group questions, but they will be more challenging and will be graded more rigorously. Bonus points are tallied separately from your normal points, and will be applied to your grade *after* the curve. Bonus points are worth more than normal points, and can have a significant positive affect on your grade. Most importantly, bonus problems are fun!

Directions for Homework 6: For all of the following dynamic programming questions, we want *all* the following information:

- State how your subproblems are defined, indexed, and what the base cases are.
- Describe the main recursive idea of your solution, with an explanation of your reasoning.
- Write pseudocode to solve the problem in polynomial time. You may choose to use either an iterative or memoized solution, unless we explicitly tell you which one to use.
- State the final running time of your algorithm, with some explanation. Faster while still correct is worth more credit. You will get more partial credit for a correct but slower solution with the proper analysis than for an incorrect, allegedly faster solution.

Example:

- CMM(i,j) is the least cost needed to multiply matrices M_i through M_j.
  The cost to multiply $M_i \dots M_i$ is 0, so $CMM(i,i) = 0$ for all i.
- "The last multiplication used to multiply the matrices from M_i through M_j is a product of the form $(M_i \cdot \dots \cdot M_k) \cdot (M_{k+1} \cdot \dots \cdot M_j)$ for some k. For any given k, the min cost is the sum of the minimum cost to evaluate each of those subproducts, which can be computed recursively, plus the cost to perform "this" multiplication, which is $M_i.width \cdot M_k.height \cdot M_j.height$. We can then iterate over all possible k where $i < k < j$ and choose the k that minimizes the total cost. This gives a final solution of

$$CMM(i,j) = \min_{k:i<k<j} CMM(i,k) + CMM(k+1,j) + M_i.width \cdot M_k.height \cdot M_j.height.$$

- Pseudocode goes here.
- We have $O(n^2)$ subproblems, and do $O(n)$ work per subproblem, for $O(n^3)$ total work.

This homework is a bit long, so start early! I also added a couple of practice problems at the end of this homework to help you to study for your test. They are optional, but are not easy, so I have labeled them "practice" to distinguish them from the simpler "optional" questions. We will post solutions to the practice problems along with your homework solutions.

1. (optional) **Class survey**
   (a) So far, how does the material covered in this class match your expectations coming in? Do you feel like your problem solving skills and toolkit are improving?

   Solution:

   (b) Do you attend office hours? If not, why? If you would like to attend office hours but cannot make the current scheduled times, what times would work for you?

   Solution:

   (c) Is there anything else that you'd like the TA's and instructor to know?

   Solution:

2. (optional) There are n rocks in a river arranged in a line, and a frog is using them to jump across. At each step, the frog can either jump to the next rock, or it can skip the next rock and land two steps ahead. It cannot skip more than one rock because that would be too far for the frog to hop. Also the frog must land on the first rock and the last rock. For example, if there are 4 rocks, 1,2,3 and 4, the frog could take the following routes: 1,2,3,4 or 1,3,4 or 1,2,4
   (a) Write an efficient *iterative* dynamic programming algorithm (do not use memorization), that takes a number of rocks and counts the number of the feasible paths that the frog can take.

Solution:

- Let $P(i)$ be the number of feasible paths from rock 1 to rock i.
- Claim: $P(i) = P(i-1) + P(i-2)$. Consider the last jump. Either the frog jumped from rock $i-1$ or rock $i-2$. So, the total number of feasible paths to rock i is the number of feasible paths to rock $i-1$ which is $P(i-1)$ plus the number of feasible paths to rock $i-2$ which is $P(i-2)$. There are two base cases, $P(1) = P(2) = 1$.

  You should be able to recognize this as the Fibonacci sequence.
- **function** FrogPaths(n)
  Array $P[1 \dots n]$
  $P[1] = 1$
  $P[2] = 1$
  for $i = 3$ to n **do**
  $P[i] = P[i-1] + P[i-2]$
  return P[n]
- The **for** loop goes from 1 to n and each iteration does $O(1)$ work. The running time is therefore $O(n)$.

(b) What if the frog can now also jump 1, 2, or 3 rocks ahead? For the previous example, the frog can now also take route 1,4. Describe how you would modify your original solution to accommodate this change. Make sure to state the new recurrence and describe the modifications to your source code.

Solution: Now there are 3 choices for the last jump. Either the frog jumped from rock $i-1, i-2$, or $i-3$. Therefore the recurrence becomes:

$$P(i) = P(i-1) + P(i-2) + P(i-3).$$

The pseudo code is modified as followed. Notice that in addition to changing the calculation for $P[i]$ a new base case is added.

function FrogPaths(n)
Array $P[1 \dots n]$
$P[1] = 1$, $P[2] = 1$, $P[3] = 2$
for $i = 4$ to n **do**
$P[i] = P[i-1] + P[i-2] + P[i-3]$
return P[n]

3. (optional) Modify the algorithm given in class for the longest common subsequence (LCS) to output both the actual longest common sequence along with the length. You don't need to do the steps of solving a dynamic program, but just show the modified pseudocode.

Solution: We will write up a complete solution for the longest common subsequence problem instead of just showing the modifications.

1. Let $LCS(i, j)$ be the *length* of the longest common subsequence between the first i characters of x and the first j characters of y.

2. We have two cases.

   - If $x_i = y_j$, then we we argue that we may as well match x_i to y_j.
     If neither x_i or y_j are matched, then we can add the match $x_i = y_j$ to the end of the substring and make it longer. If only one of $x_i = y_j$ are matched, say without loss of generality that x_i is matched, then since x_i is the last character in x, it matches last matched charcter in y. We can switch $x's$ match with this character to a match with y_j without changing the length of the total match. Thus, we may as well match x_i to y_j.
     Therefore, the LCS will have this final character $x_i = y_j$ appended to the end of whatever the longest common subsequence was between $x[1 \cdots i-1]$ and $y[1 \cdots j-1]$.
   - If $x_i \neq y_j$, then these characters cannot match each other. Thus at least one of these isn't matched to any character, and by removing that unmatched character, we do not change the LCS. In this case, the LCS(i,j) is one of LCS(i-1,j) or LCS(i,j-1). Naturally, it is the larger of the two.

   Therefore, the recurrence is $LCS(i, j) = 1 + LCS(i-1, j-1)$ if $x_i = y_j$, and $LCS(i, j) = \max(LCS(i-1, j), LCS(i, j-1))$ otherwise.

3. Now, we compute these recurrences and store the computed values in an array. Sometimes in dynamic programming algorithms, it helps to store some auxilary data in a separate array when reconstructing the final answer. In this example, I will use only the array for the lengths without storing extra information. In future problems, I will store extra information.

```
function LCS(String X[1...m], String Y[1...n])
    Array A[0...m × 0...n] initialized to 0 ▷ We really only need to initialize the 0 row and column
    for i = 1 to m do
        for j = 1 to n do
            if X[i] = Y[j] then
                A[i, j] ← A[i − 1, j − 1] + 1
            else
                A[i, j] ← max(A[i − 1, j], A[i, j − 1])
                        ▷ At this point, the length of the LCS is stored in A[n, m]
    i ← m
    j ← n
    Answer ← ”                ▷ an empty linked list of characters to store our answer
    while i > 0 & j > 0 do
        if X[i] = Y[j] then
            Answer = X[i] + Answer                ▷ prepend the matched character
```

```
            i ← i − 1
            j ← j − 1
        else if A[i − 1, j] > A[i, j − 1] then
            i ← i − 1
        else
            j ← j − 1
    return Answer
```

Note that in recovering the solution, we basically recompute some of the same work that we did in creating the array A. This is why it's usually better to store some auxillary information when doing the main for loop to make recovering the solution faster.

4. Computing the array A does a constant amount of work per entry, and there are $(m+1)(n+1)$ entries, for a grand total of $O(mn)$ work.

   Recovering the solution from the array is done with a while loop, and does a constant amount of work per step. Each step decreases the value of one of i or j, and we quit when either of them is 0. Thus we will terminate in at most $O(m+n)$ steps, for a total of $O(m+n)$ additional work done to recover the solution.

4. (group) Yuckdonald's is considering opening a series of restaurants along Quaint Valley Highway (QVH). There are n viable locations along this highway, and you are given the mile markers of these locations in a array $M = [m_1, m_2, \ldots, m_n]$ that is already sorted.

   - At each location, Yuckdonald's may open at most one restaurant. The profit from opeing a restaurant at location i is $p_i > 0$. You are given input array $P = [p_1, p_2, \ldots]$
   - Any two restaurants must be at least k miles apart, where k is an input positive integer.

   Give an efficient **memoized** algorithm to compute the maximum possible profit, AND **return a list of the locations that I should open**.

Solution: We are computing the maximum profit (not return the list of stores to open), so our algorithm will not worry about storing the best sequence of restaurants to open.

1. Let $BestProfit(i)$ be the most profit you can make among the first i possible restaurants.

2. We have two cases for what to do with restaurant i. We could potentially leave it closed, in which case we are looking at the best profit to be made among the first $i-1$ restaurants. In this case, the best profit would be $BestProfit(i-1)$.

   If we do open the ith restaurant, then you can't open any restaurants between $m_i - k$ and m_i. Let j_i be the index of the closest restaurant to i that is that least k miles away from i. In otherwords, let $j_i < i$ be the largest possible index such that $m_{j_i} + k \leq m_i$. Then the best profit would be the profit from restaurant i, plus the best possible profit among the first j_i restaurants. In this case, the profit is $p_i + BestProfit(j_i)$. These are

the only two options (opening or closing i), so we choose the better of these two options, $BestProfit(i) = \max(BestProfit(i-1), p_i + BestProfit([\max_{j<i} |m_j < m_i - k]))$.

3. At each step, we need to find $\max_j |m_j < m_i - k$. We could do this using a binary search, which this adds $O(\log(n))$ work to each step, and would result in an $O(n \log n)$ algorithm. Instead, we will precompute ALL the values of j_i for each i in $O(n)$ total time before hand, store it in an array J, then use this value of $J[i]$ in the algorithm. This will result in an $O(n)$ total algorithm - $O(n)$ for precomputation and $O(n)$ for the dynamic program.

   (We could have also computed all the values of j on the fly inside the main recursive loop, but this will be easier to follow).

   We'll set $J[i] = null$ to mean that no restaurant is available k miles behind i (in otherwords, i is less than k miles in front of the first restaurant). For convenience, we will also let $J[0] = 0$.

   Since we have the preprocessing step outside of the main recursive loop, we will have two functions here. We'll have an outside "main" function that does the preprocessing and that calls the recursive function.

```
function MAIN(Array m[1...n], Array p[1...n], k)
    Array J[0...n]                ▷ J[i] will store the closest resaurant k miles behind.
    J[0] ← 0
    for i = 1 to n do
        J[i] ← J[i − 1]                                  ▷ Start at the previous value
        while m[j[i]] < m[i] − k do
            J[i] ← J[i] + 1
        J[i] ← J[i] − 1                  ▷ We will increment once too many times
                                         ▷ After this, J[n] has been filled
    Array A[0...n] init to Null          ▷ Here we begin the main dynamic program
    function BESTPROFIT(i)                                ▷ Recursive Part Goes Here
        if A[i] = Null then
            if i = 0 then
                A[i] ← 0
            else
                A[i] ← max(BestProfit(i − 1), BestProfit(J[i]) + p[i])
        return A[i]
    return BestProfit(n)                                  ▷ This is what the main function calls
```

   Here's the same idea presented as an iterative dynamic program.

```
function BESTPROFIT(Array m[1...n], Array p[1...n], k)
    Array J[0...n]                ▷ J[i] will store the closest resaurant k miles behind.
    J[0] ← 0
    for i = 1 to n do
        J[i] ← J[i − 1]                                  ▷ Start at the previous value
        while m[j[i]] < m[i] − k do
            J[i] ← J[i] + 1
        J[i] ← J[i] − 1                  ▷ We will increment once too many times
```

▷ After this, $J[n]$ has been filled
Array $A[0 \dots n]$ ▷ Here we begin the main dynamic program
$A[0] \leftarrow 0$
for $i = 1$ to n **do**
$A[i] \leftarrow \max(A[i-1], A[J[i]] + p[i])$
return A[n]

4. We have two for loops that each do $O(n)$ work, which means we do a total of $O(n)$ work.

   Note: The reason that the precomputation step only does $O(n)$ work is that the value of $J[i]$ in the while loop only increases from 1 to n, although sporadically. Since it only increases, it can only do so at most n times.

5. Now, we modify our dynamic program that outputs the maximum *profit* to also return the list of restaurants I should open.

   Like the examples done in class, what we need to do is store the choices that we make. In this case, the choice was to open or not to open this restaurant, so we can use a separate boolean array called $Open[i]$ that stores this choice as a 1 if we open, or 0 if not.

   The following is conceptually important. Note that we can't run through the $OPEN$ array at the end and output all the restaurants that we marked 1. The set of restaurants that we mark open are NOT the set of all restaurants that you should open! We marked them OPEN because that was the right thing do to when looking at only the restaurants $1 \cdots i$, which is not necessarily the right thing to do for $1 \cdots n$.

   It may help you to do a trial run to understand why this is different. Here is the modified psuedocode for the memoized version:

function MAIN(Array $m[1 \dots n]$, Array $p[1 \dots n]$, k)
Array $J[0 \dots n]$
$J[0] \leftarrow 0$
for $i = 1$ to n **do**
$J[i] \leftarrow J[i-1]$
while $m[j[i]] < m[i] - k$ **do**
$J[i] \leftarrow J[i] + 1$
$J[i] \leftarrow J[i] - 1$
Array $A[0 \dots n]$ init to Null
Array $Open[0 \dots n]$ ▷ $Open[i] = 1$ iff we open i in the subproblem $1 \cdots i$
function BESTPROFIT(i) ▷ Recursive Part Goes Here
if A[i] = Null **then**
if i = 0 **then**
$A[i] \leftarrow 0$
$Open[0] \leftarrow Null$
else
if $BestProfit(i-1) > BestProfit(J[i]) + p[i]$ **then**
$A[i] \leftarrow A[i-1]$ ▷ This could also have been
$A[i] \leftarrow BestProfit(i-1)$, but I called it earlier, so it's already been assigned.

```
            Open[i] ← 0                                   ▷ The new part
        else
            A[i] ← A[J[i]] + p[i]
            Open[i] ← 1                                   ▷ The new part
    return A[i]
Answer = BestProfit(n)                                    ▷ Call the recursive function
while curnode > 0 do                                      ▷ Here we assemble the list to return
    if Open[i] = 1 then
        FinalList ← [curnode] + FinalList                 ▷ prepend the previous node
        curnode ← J[curnode]
    else
        curnode ← curnode − 1
return Answer, FinalList
```

And here's the modified pseudocode for the iterative version.

```
function BestProfit(Array m[1...n], Array p[1...n], k)
    Array J[0...n]
    Array Open[0...n]          ▷ Open[i] = 1 iff we open i in the subproblem 1 ··· i
    J[0] ← 0
    Open[0] ← Null
    for i = 1 to n do
        J[i] ← J[i − 1]
        while m[j[i]] < m[i] − k do
            J[i] ← J[i] + 1
        J[i] ← J[i] − 1
    Array A[0...n]
    A[0] ← 0
    for i = 1 to n do
        if A[i − 1] > A[J[i]] + p[i] then
            A[i] ← A[i − 1]
            Open[i] ← 0                                   ▷ The new part
        else
            A[i] ← A[J[i]] + p[i]
            Open[i] ← 1                                   ▷ The new part
    FinalList ← [ ]
    curnode ← n
    while curnode > 0 do                                  ▷ Here we assemble the list to return
        if Open[i] = 1 then
            FinalList ← [curnode] + FinalList             ▷ prepend the previous node
            curnode ← J[curnode]
        else
            curnode ← curnode − 1
    return A[n], FinalList
```

We assumed that you can do the prepending step in $O(1)$ time, which you can do in a

linked list.

5. (group) A certain pipe-processing plant has a machine that can cut a piece of pipe into two peices at any position. This operation incurs a cost equal to the length of pipe being cut, n, regardless of the location of the cut. So to cut a 5 meter peice of pipe costs \$5, whether you cut it into 1 and 4 meter peices or 2 and 3 meter peices.

   You are given a very long, n meter length of pipe, and have marked on that pipe the places where you need to make cuts. The order in which the breaks are made can affect the total cost. For example, if you want to cut a 20-foot piece at positions 3 and 10, then making the cuts at position 3, then at 10 incurs a total cost of $20 + 17 = 37$, while doing position 10 first has a better cost of $20 + 10 = 30$.

   Give an **iterative** dynamic programming algorithm that, given the locations of $m - 1$ cuts in a piece of pipe of length n, finds the minimum **cost** of breaking it into m pieces. For convenience, you may want to assume that the input array is of the form $[c_0, c_1, \ldots, c_{m-1}, c_m]$, where $c_0 = 0$ and $c_m = n$. Think carefully about how to define your subproblems, and your final runtime shouldn't be much longer if all lengths were multiplied by 1000.

Solution:

1. The general setup is similar to that of chain-matrix multiplication. Lets say the locations of the m cuts were given to you in an array l.

   Let $CutCost(i, j)$ be the minimum cost of making the $j - i - 1$ remaining cuts on the peice of pipe between cuts i and j, if you were given that as one solid piece with the ends already cut off. For notational convenience, if treat positions $l[0] = 0$ and $l[m] = n$ like cuts in our array, then the value we are looking for is $CutCost(0, m)$.

2. Given a segment between cuts i and j, we choose which cut to make first, and we incur a total cost of $l[j] - l[i]$ no matter where we make the cut. What we will be left with is two peices of pipe that we may then have to cut further.

   We might have some intuition about where this first cut should probably be (near the middle, in some sparse area, etc.), but we must ignore that intuition, because the night is dark and full of counterexamples. So to avoid potentially misleading intuition, we instead check all possible choices of the next cut, and choose the best one.

   For any choice of cut x, where $i < x < j$, we have two pieces left, one from i to x, and another from x to j. To pick out the best one, then $CutCost(i, j) = (l[j]-l[i])+\min_{i<x<j}(CutCost(i, x)+CutCost(x, j))$. Note that to solve $CutCost(i, j)$, we need to know values of $CutCost$ solely on subproblems with a smaller difference between the indices. Thus, in our for loop, we will solve all subproblems of a particular distance between indices before increasing. Also, the base cases can be $CutCost[i, i + 1] = 0$, since that piece of pipe no longer needs to be cut.

3. **function** CutCost(Array $l[0 \ldots m + 1]$)
   Array $CC[0 \ldots m + 1 \times 0 \ldots m + 1]$

for $i = 0$ to m **do**
$CC[i, i+1] \leftarrow 0$
▷ Initialization of values
for $l = 2$ to $m+1$ **do** ▷ The increasing distance between i and j
for $i = 0$ to $m+1-l$ **do**
$CC[i, i+l] \leftarrow l[j] - l[i] + \min_{i<x<j}(CC[i, x] + CC[x, j])$
return $CC[0, m]$

We could have also done the iteration in a different way (see the chain matrix multiplication example).

4. We have $O(n^2)$ entries in our array and do $O(n)$ work to solve each entry, resulting an in $O(n^3)$ algorithm.

6. (group) Given two sequences X and Y, let $C(X, Y)$ denote the number of times that X appears as subsequence of Y. For instance, the sequence AB appears 5 times as a subsequence of $ADABACB$. Write an efficient dynamic programming algorithm to calculate $C(X, Y)$. (Hint: Consider the solutions to prefixes of the input as your subproblem.)

Solution:

- Let $C(i, j)$ be the number of occurences of the first i characters of X as a subsequence in the first j characters of Y.

  Let X_i denote the first i characters of the string X and let $X[i]$ denote the i^{th} character (similarly for Y). Let m denote the length of X and let n denote the length of Y.

- If the last characters of X_i and Y_j don't match, then the last character of Y_j is not involved in any occurence of X_i, so the count is the same as the number of occurences of X_i in Y_{j-1}. If the last characters do match, then we also include the number of occurences of X_i that involve matching $X[i]$ with $Y[j]$, namely $C(i-1, j-1)$, plus the number that don't, $C(i, j-1)$. So the recurrence is: $C(i, j) = C(i, j-1)$ if $X[i] \neq Y[j]$, and $C(i, j) = C(i, j-1) + C(i-1, j-1)$ if $X[i] = Y[j]$.

  Now we set up the base cases so that your recurrence is correct for small values. $C(X_0, Y_j) = 1$, and $C(X_i, Y_0) = 0$ if $i \geq 1$.

- **function** NumSubsequence(X, Y)
  Array $C[0 \ldots m, 0 \ldots n]$
  for $i = 1$ to m **do**
  $C[X_i, Y_0] = 0$
  for $j = 0$ to n **do**
  $C[X_0, Y_j] = 0$
  for $i = 2$ to m **do**
  for $j = 2$ to n **do**
  if $X_i = Y_j$ **then**
  $C[X_i, Y_j] = C[X_i, Y_{j-1}] + C[X_{i-1}, Y_{j-1}]$
  else

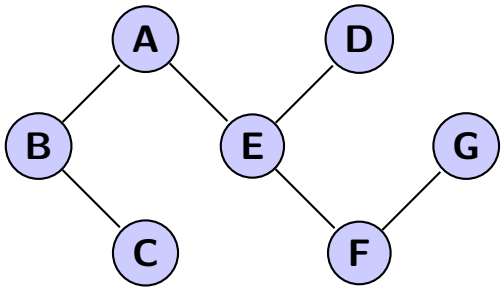
$C[X_i, Y_j] = C[X_i, Y_{j-1}]$
return $C[X_m, Y_n]$

- The running time is $O(mn)$. We have $O(mn)$ entries in our table and it takes $O(1)$ time to compute each entry.

7. (group) A *vertex cover* of a graph $G = (V, E)$ is a subset of the vertices $S \subset V$ that includes at least one endpoint of every edge in E. Give a linear-time dynamic programming algorithm for the following task.

   Input: An undirected tree $T = (V, E)$.
   Output: The **size** of the smallest vertex cover of T.

   For instance, in the following tree, possible vertex covers include $\{A, B, C, D, E, F, G\}$ and $\{A, C, D, F\}$ but not $\{C, E, F\}$. The smallest vertex cover has size 3: $\{B, E, G\}$.



Solution: Assume that the tree is rooted at some node r, otherwise pick one at random to be the root.

There are multiple possible solutions for this problem (at least 4)!

1. Let $VCover(n)$ be minimum cost of a vertex cover of the subtree descending from node n. For a leaf, we set $VCover(n) = 0$.

2. At any node n, we have the choice of including the vertex n, or not including the vertex n. If we include n, we cover all the edges between n and its children. The remaining subtrees descending from n's children still need to be covered, in which case $VCover(n) = 1 + \sum_{x \in n.children} VCover(x)$.

   If we do not include n, then we MUST include every child of n in the vertex cover. This leaves all the subtrees descending from n's gradchildren uncovered. In this case, $VCover(n) = |n.children| + \sum_{x \in n.children} \sum_{y \in x.children} VCover(y)$. We choose the smaller of the two.

3. This is a memoized-ish approach, since we look at children and grandchilden, this is easier. If we used a solution that only every looked at children, then we could have done this in a more tree-search style implementation.

   Global Tree T

function INIT(node n) ▷ Initializing takes $O(|V|)$ time $n.cover \leftarrow NULL$
  for $c \in n.children$ **do**
    $Init(c)$
function VCOVER(node n)
  if $n.cover = NULL$ **then** $n.cover \leftarrow \min(1+\sum_{x \in n.children} VCover(x), |n.children|+ \sum_{x \in n.children} \sum_{y \in x.children} VCover(y)$
  return n.cover
return $Init(T.root)$.
return $VCOVER(T.root)$.

4. Outside the summations, we do $O(1)$ work per node.

   Inside the summations, at each node n, we do an amount of work proportional to the number of all n's children and all n's grandchildren. This could be large for some nodes and small for others. To count this a different way, think about it backwards.

   Any node n is only ever considered inside a summation by n's parent or n's grandparent. This means $VCover(n)$ is only ever called inside a summation $O(1)$ times per node. Thus, the total work done by all the summations over every call of $VCover$ is $O(|V|)$.

   This gives us a total running time of $O(|V|)$. (Recall that in a tree, $|V| = |E| + 1$).

There is an alternate solution that keeps track of two values of $VCOVER$ per vertex, one where you do include the vertex and one where you don't.

There is a third greedy solutions that looks at leaves and works it's way up.

There is a fourth solution that makes use of a connection between independent sets and vertex covers.

You may notice that there are some strong similarities between the independent set algorithm given in class and the vertex cover algorithm here. There's a reason that's clear if you think about it from the perpective of the edges. In an independent set, every edge has 0 or 1 of its end points included in the set, but not 2. In a vertex cover, every edge has 1 or 2 edges included in the set, but not 0. Thus the *complement* of an independent set I is in fact a vertex cover with $n - |I|$ elements, and vice versa! Furthermore the complement of a *maximum* independent set must be a *minimum* vertex cover!

Thus the following is an acceptable solution, if accompanied by something like the above proof of correctness.

"Run the maximum independent set algorithm given in class. Return the complement set of vertices."

8. (bonus) You are a brick layer, and you are trying to build a wall out of bricks. Each brick i has height 1 but can have variable length l_i. At every step, someone hands you a brick and you have the option of either adding it to the "current" layer of bricks, or starting a new layer of bricks on top of the old layer. Once you start a new layer of bricks, you cannot add bricks to lower layers. Naturally, every layer needs to be at least as long as the one above it.

   Your goal is to make this wall as tall as possible.

   Give an efficient algorithm for finding the way to make a maximum height wall if you must

follow a predetermined sequence of bricks. (You know the sequence of lengths in advance, but you cannot change it). More bonus points are given for faster (correct) algorithms.

Hint: You can probably go faster, but you may need to "hack" your dynamic programmic solution to get there.

Solution: There are a few key points that make this possible in $O(n)$.

- Your supbroblems should start from the back, i.e. consider laying the bricks from $i \dots n$.
- Finding the max height that this subproblem can reach isn't enough; sometimes there are equivalent ways to get to the same height, but some of them have different lengths at the bottommost layer.
- Minimizing the length of the bottommost layer *IS* enough; the configuration with smallest bottommost layer (and smallest secondmost bottom layer among those with smallest bottommost layer, etc.) will have maximum height; we'll prove this later.
- Dealing with the min length instead of the max height leads to a natural $O(n^2)$ dynamic programming algorithm. We can "hack" this into an $O(n)$ algorithm with an observation about how one of the indices behaves.

Lemma 8.1: If $j \geq i$, $BestHeight(i) \geq BestHeight(j)$

PROOF: This is true because we can always add any bricks $i, i+1, \dots j-1$ to the bottommost level of any optimal configuration for $j \dots n$ while retaining the same height. ■

Lemma 8.2: If the length of the bottommost level of the wall is minimized, and subject to this restriction the second bottommost level is minimized, etc., then this wall has maximal height.

PROOF: We prove this by induction. In the base case, we have only one brick, and the correctness of this statement is trivial.

Say the theorem is true for all $x \dots n$ for $x > i$. The wall with minimal width at the bottommost level also leaves a maximal number of bricks for the layers above. By 8.1, it follows that the best height acheivable by the remainder of the bricks is maximal over all other possible towers. By the induction hypothesis, because the remainder of the bricks also follows the minimum base width principle, the remainder of the bricks in fact attains this maximal height. Therefore, the tower in its entirety attains maximal height. ■

1. Let $MinBase(i)$ be a tuple containing the minimum base length obtained from laying bricks $i \dots n$. So, $MinBase(n) = l_n$. We will concurrently store the $BestHeight(i)$ to be the best height from laying bricks $i \dots n$. $BestHeight(n) = 1$. We'll use the convention that $MinBase(n+1) = BestHeight(n+1) = 0$ to make the recursion cleaner.
2. Based on the lemma, we want the base to be as small as possible. Thus, to calculate $MinBase(i)$, we need the calculate the smallest value of j such that $MinBase(j) \leq \sum_{x=i}^{j-1} l_x$. Then, $MinBase(i) = \sum_{x=i}^{j-1} l_x$, $BestHeight(i) = BestHeight(j) + 1$.

3. Turning the above ideas into an $O(n^2)$ algorithm, we would have:

```
function BrickLaying(Array i[1...n])
    Array MinBase[1...n+1]
    Array BestHeight[1...n+1]
    MinBase[n+1] ← 0
    BestHeight[n+1] ← 0
    for i = n to 1 do
        j ← i+1
        sum ← l_i
        while MinBase[j] > sum do
            sum = sum + l_j
            j = j + 1
        MinBase[i] = sum
        BestHeight[i] = BestHeight[j] + 1
                                          ▷ Initialization of values
    return BestHeight[1]
```

   This algorithm does a loop with n iterations, and does $O(n)$ work per iteration in the while loop, which leads to an $O(n^2)$ algorithm.

4. We now optimize the way that we find the index j at each step. One key observation is that j never "retreats"! That is, if for some value of i, $MinBase(j) \leq \sum_{x=i}^{j-1} l_x$, then this will also be true for any smaller value of i (because you are adding more positive things!). So, we avoid starting our search for the right j "from scratch" for every i. This is a lot like the precomputation step in the Yuckdonalds problem - but we are doing it concurrently with the main for loop instead of outside the loop in a preprocessing step. Instead of increasing j until we have a feasible placement, we decrement j until we no longer have a feasible placement, and keep the same value of j between iterations.

```
function BrickLaying(Array i[1...n])
    Array MinBase[1...n+1]
    Array BestHeight[1...n+1]
    MinBase[n+1] ← 0
    MinBase[n] ← l_n
    BestHeight[n+1] ← 0
    BestHeight[n] ← 1
    j ← n+1
    sum ← l_n
    for i = n-1 to 1 do
        sum ← sum + l_i
        while MinBase[j-1] ≤ sum - l_{j-1} do
            sum = sum - l_{j-1}
            j = j - 1
        MinBase[i] = sum
        BestHeight[i] = BestHeight[j] + 1
    return BestHeight[1]
```

This algorithm does a loop with n iterations, and does constant work per iteration, outside of the while loop. The value j will increment at most once over every value of j over the course of the entire while loop (though it might do so unevenly as i increments), but there is a total of $O(n)$ work done total by the while loop. Overall, this leads to an $O(n)$ algorithm.

9. (practice) For strings $X = x_1 \dots x_n$, $Y = y_1 \dots y_m$ and $Z = z_1 \dots Z_{m+n}$, we say that Z is an *interleaving* of X and Y if it can be obtained by interleaving the bits in X and Y in a way that maintains the left-to-right order of the bits in X and Y. For example, if $X = abc$ and $Y = dcab$ then $Z = x_1x_2y_1x_3y_2y_3y_4 = abdccab$ is an interleaving of X and Y whereas $acdcbab$ is not. Give the most efficient *dynamic programming* algorithm you can to determine if Z is an interleaving of X and Y or not. Prove your algorithm is correct and analyze its time complexity as a function of $n = |X|$ and $m = |Y|$.

Solution:

- Let $I(i, j) = TRUE$ if $z_1 \dots z_{i+j}$ is an interleaving of $x_1 \dots x_i$ and $y_1 \dots y_j$ and $FALSE$ otherwise.
- The resursion depends a lot on the relationship between the characters x_i, y_j and z_{i+j}. So we identify the following cases.
  - Case 1: $z_{i+j} = x_i, z_{i+j} \neq y_j$
    In this case, the last character of $z[i + j]$ could only have come from $x[i]$. Thus z is an interleaving of $x[1 : i]$ and $y[1 : j]$ if and only if $z[i + j - 1]$ is an interleaving of $x[1 : i - 1]$ and $y[i : j]$. Thus, $I(i, j) = I(i - 1, j)$.
  - Case 2: $z_{i+j} = y_j \neq x_i$
    Similar to the above case, $I(i, j) = I(i, j - 1)$.
  - Case 3: $z_{i+j} = x_i = y_j$
    In this case, the final character could come from either side. In this case, $I(i, j) = I(i, j - 1) \vee I(i - 1, j)$.
  - Case 4: if $z_{i+j} \neq x_i, z_{i+j} \neq y_i$
    If the final character doesn't match either side, then neither character could correspond to the last character for z. In this case, $I(i, j) = False$.
- Now to write this as psuedocode,

```
Global Array I[1...n × 1...m] initialized to Null
function INTERLEAVE(I,J)
    if i < 0 or j < 0 then
        return False
    if I[i, j] is Null then
        if i = 0 and j = 0 then
            I[0, 0] ← True
        else
            I[i, j] ← False
```

if $Z[i+j] = X[i]$ **then**
$I[i,j] \leftarrow Interleave(i-1,j)$
if $Z[i+j] = Y[j]$ **then**
$I[i,j] \leftarrow I[i,j] \vee Interleave(i,j-1)$
return $I[i,j]$

- The array has at most nm entries and we do $O(1)$ work per entry, thus this takes $O(nm)$ time.

10. (practice) Gven a set of denominations $x_1 < x_2 < \ldots < x_n$, we wish to make change for a value v. We saw in class that a greedy approach may not always use the fewest possible coins.

    Write an $O(nv)$ *iterative dynamic programming* algorithm (do not use memorization), that takes the denominations in array X and returns the smallest number of coins needed to make chance for value v if it is possible, and "Null" if it is impossible.

Solution:

- Let $Change(i,t)$ be the minimum number coins needed to make change for a value of t using the first i coins, or $Null$ if it is impossible.

  Your base cases would be 0 if $t = 0$, and *null* if $t < 0$, and also null if $i = 0$ but $t > 0$.

- We should decide if the best way to make change for t using the first i coins should use a coin with value x_i or not. If it does, then the total number of coins we need is one more than the number needed to get the value $t - x_i$. If it doesn't, then this is a same way to make t using the first $i-1$ coins. Thus, $Change(i,t) = \min(Change(i-1,t), 1 + Change(i,t-x_i))$.

- **function** CHANGE(A,v)
  Array $S[n \times (v+1)]$
  for $i = 0$ to n **do**
  $S[i,0] \leftarrow 0$
  for $t = 1$ to v **do**
  $S[0,t] \leftarrow False$ ▷ Base Case Initialization
  for $i = 1$ to n **do**
  for $t = 0$ to v **do**
  $S[i,t] \leftarrow S[i-1,t]$
  if $x_i \leq t$ **then**
  $S[i,x] \leftarrow \min(S[i,x], S[i-1,t-x_t]+1)$
  return $S[n,v]$

- We have $n \times v$ entries in our array, each of which is computed in $O(1)$ time, which leads to an $O(nv)$ algorithm.

11. (practice) A subsequence is a *palindromic* if it is the same whether read left to right or right to

left. For instance, the sequence

$$A, C, G, T, G, T, C, A, A, A, A, T, C, G$$

has many palindromic subsequences, including A, C, G, T, G, C, A and A, A, A, A (on the other hand, the subsequence A, C, T is *not* palindromic). Devise an algorithm that takes a sequence $x[1 \dots n]$ and returns the **length** of the longest palindromic subsequence. Its running time should be $O(n^2)$.

Solution:

1. Let $Palin(i, j)$ be the length of the longest palindromic subsequence between (and including) characters i and j. We will use the convention that $Palin(i, i-1) = 0$ (a zero-length string), and that $Palin(i, i) = 1$, since an individual character is a palindrome.

   You could have chosen a different workable solution with a different convention for the base cases.

2. This is a bit like longest common subsequence. Let x be our string. Then if $x[i] = x[j]$, then these endpoints can match with each other, and the length of the longest palindrome is $1 + Palin(i+1, j-1)$.

   If $x[i] \neq x[j]$, then these endpoints are not both in the longest palindrome, and we may eliminate one of them without changing the longest palindrome. We don't know for sure if either of them are used in the longest palindrome so far, we just know it can't be both. Therefore, we should choose $\max(Palin(i+1, j), Palin(i, j-1)$ to cover both cases. Like in chain matrix multiplication or the pipe cutting problem, we can compute these values in order of the increasing *length* of the sequence.

3. ```
   function LongestPalin(Array x[1...n])
       Array Palin[1...n × 0...n]
       for i = 1 to n do
           Palin[i, i] ← 1
           Palin[i, i-1] ← 0
                                          ▷ Initialization of values
       for l = 1 to n - 1 do              ▷ Increasing lengths
           for i = 1 to n - l do
               j ← i + l
               if x[i] = x[j] then
                   Palin[i, j] ← 1 + Palin[i + 1, j - 1]
               else
                   Palin[i, j] ← max(Palin[i + 1, j], Palin[i, j - 1])
       return MaxL
   ```

4. Each entry of the array takes constant work, and we compute values for about half of the $n(n+1)$ entries, which gives us an $O(n^2)$ algorithm in total.
   ```