CSCI 232 Program 4

Due Sunday June 25th @ 11:59 PM. Please submit this assignment (.java files) to the appropriate dropbox on D2L.

Yeah, so we are not doing the closest pair problem for program 4. It was a bit too difficult, and I don't think we have enough time for me to give it to you as an assignment.

Instead, this assignment will focus on Dynamic Programming. We will likely write some, or all of the code for this assignment on Wednesday June 20th. This is about the length of a lab, but because this lab involves recursion, it will probably be a bit more difficult.

Background

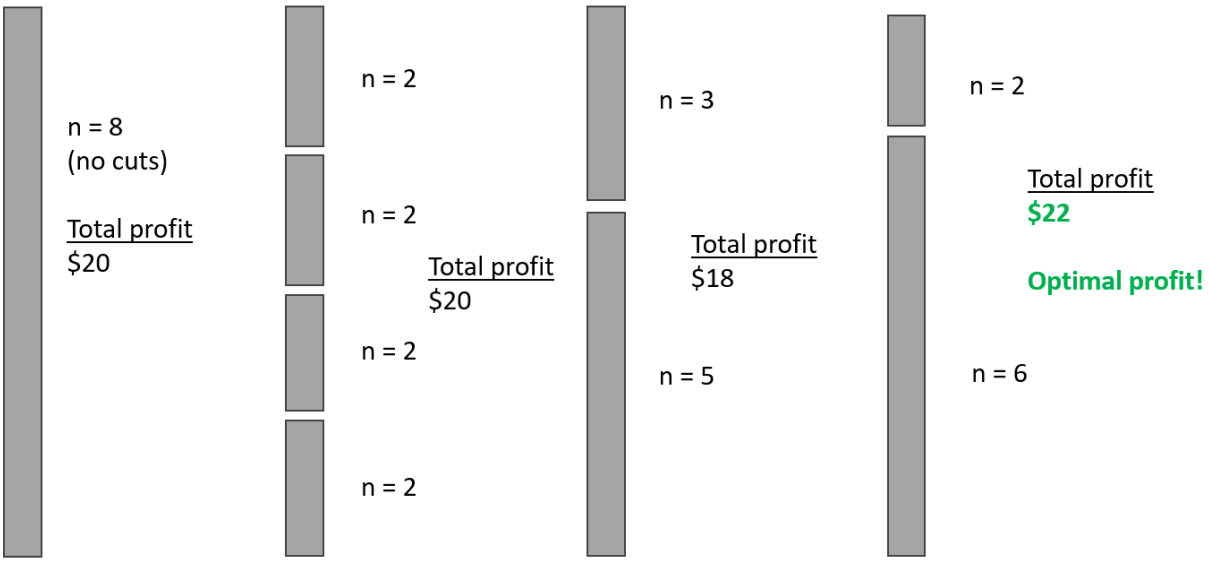
In this assignment, you are going to write a program that solved the rod cutting using dynamic programming. We discussed the rod cutting problem during class on Tuesday June 19th, but here is the gist of it:

Given a rod of length n inches, and an array of prices that includes prices of all pieces of size smaller than n, determine the maximum value obtainable by cutting up the road and selling the pieces.

Suppose we have a rod of length 8, and the following price table:

Length	1	2	3	4	5	6	7	8
Price	1	5	8	9	10	17	17	20

There are many ways to cut this rod, but the maximum obtainable value is 22 (a cut of length 2 and 6)



(See next page for dynamic programming overview)

Instructions

Download Program4Demo.java as a starting point, you will need to write the recursive method, **`cutRod()`** that will compute the optimal cost we can get from cutting an N inch rod. Your program does not need to return which cuts were used—it only needs to return the maximum profit.

Your program **must** use recursion to compute the maximum profit. With a basic recursive approach, we will calculate all possible permutations of the different ways to cut the road, and its profit. Because recursion is being used, the same sub-problem (such as **`cutRod(3)`** will be computed many times during execution), which will give us very bad time complexity of **`O(n!)`**.

To get around this, we can use **dynamic programming**. We will use **memoization** to prevent our program from solving the same sub-problem multiple times. Implementing memoization is simply just adding an additional data structure. In our case, this will be a 2D array called **`dp[][]`**.

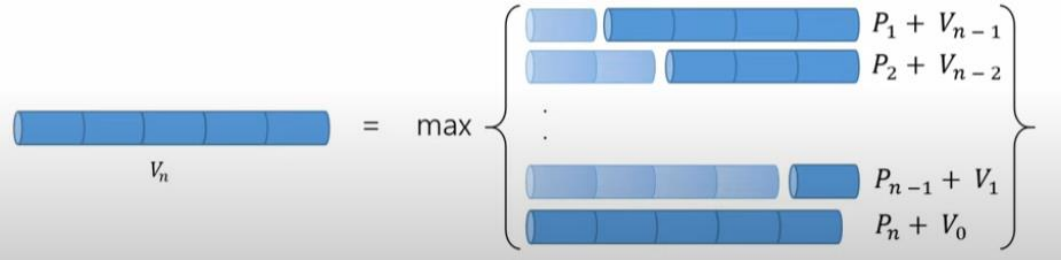
You do not need to add any other classes, and you don't need to write any other methods. You only need to fill in the body of the **`cutRod()`** method.

Algorithm Overview

At any point during the recursion stack, we will always be evaluating a potential cut to make. We always have two options. The first option is to not cut the rod anymore, so call don't cut the rod, but check the next cut value. The second option is cut the rod. However, there are many ways to cut the rod, so we will check each possible cut. Call the method again and decrease the rod length by the cut, and return the profit made by making the cut. During each recursive call, we want to return a maximum between the profit of not cutting, and cutting the road. Whenever we solve a subproblem, we need to store the solution inside our dynamic programming data structure.

Here is the problem, stated as a recurrence relation, which is easier the visualize in the image below:

$$V_n = \max_{1 \le i \le n} (P_i + V_{n-i})$$



prices[] = array that holds profits for cutting rod of length 1 to N

index = the current cut value we are checking

rod_length = length of the rod we are checking (also called N)

dp[][] = dynamic programming memoization table. Holds solutions to all subproblems

```
function cutRod(int prices[], int index, int rod_length, int[][] dp)
      if the index is 0:
            //then we have a cut length of one inch, and we cant go any smaller
            return the current length of the rod * price[0]
      if the answer to this subproblem is in our DP array:
            //we don't need to do addition work
            return dp[index][rod_length], which is which our solution should be held at

      //otherwise, we now need to do some additional recrursion
      // We can either cut the rod, or not cut the rod
      notCut = cutRod(prices, index-1, rod_length, dp)
      cut = -INFINITY
      cut_to_try = index + 1
      if  cut_to_try < rod_length:
            // make the cut (rod_length – cut_to_try), but also return the profit made
            // from piece we cut off
            cut = prices[index] + cutRod(prices, index, rod_length – cut_to_try, dp)
      //store in array, so we don't need to do it later
      dp[index][rod_length] = max(cut, not_cut)
      return max(cut, not_cut)
```

Sample Output

There are two test cases in the demo class. When you run your program, it should come up with the same solution as below:

```
Maximum profit: 22
Maximum profit: 55
```

Starting Code:

- Program4Demo.java (https://www.cs.montana.edu/pearsall/classes/summer2023/232/programs/csci232-Program4Demo.java)

Grading

• Your program correctly computes the maximum profit of the rod cutting problem using dynamic programming – **90 points**
• Each method of the program is commented – **10 points**

NOTE: If your code does not compile, correctness cannot be verified, and you won't receive any points for your code. Turn in code that compiles!