Computer Science 385
Design and Analysis of Algorithms
Siena College
Spring 2018

Lab 9: Dynamic Programming

Due: Start of your next lab session

You will be assigned a partner to work with on this lab. Only one submission per group is needed.

Group members: ____

Learning goals:

- To be able to write top-down recursive solutions to optimization problems
- To understand the dynamic programming algorithm design technique
- To be able to implement top-down dynamic programming solutions

---

Getting Set Up

You will receive an email with the link to follow to set up your GitHub repository, which will be named `dynamic-lab-yourgitname`, for this Lab. One member of the group should follow the link to set up the repository on GitHub, then that person should email the instructor with the other group members' GitHub usernames so they can be granted access. This will allow all members of the group to clone the repository and commit and push changes to the origin on GitHub.

Submitting

Once all written items are initialed to indicate completion, turn in one copy of this handout. Be sure names of all group members are clearly on the first page.

Your submission requires that all required deliverables are committed and pushed to the master for your repository's origin on GitHub. That's it! If you see everything you intend to submit when you visit your repository's page on GitHub, you're set.

Grading

Score: $\boxed{}$ / 100

We begin with a favorite problem from Discrete Math: the Chicken McNuggets problem. At a fast food restaurant, small pieces of chicken are sold in boxes. The small box contains 6 pieces, the medium box contains 9 pieces and the large box contains 20 pieces. The restaurant has an unlimited supply of the three types of boxes.

? Question 1:
Can an order for exactly 21 chicken pieces be filled using a combination of the three box sizes? If so, how? (4 points)

? Question 2:
Can an order for exactly 22 chicken pieces be filled using a combination of the three box sizes? If so, how? (4 points)

? Question 3:
Study the top-down, exhaustive search implementation below that returns true if it is possible to fill an order of the indicated size using boxes of size 6, 9, and 20. Fill in the three blanks with code to complete the implementation. (4 points)

```
ALGORITHM FillOrder($orderSize$)
    //Input: $orderSize$, the total number of nuggets desired
    //Output: true if the order can be filled, false if not
    if $orderSize = 0$ then
        return true
    if $orderSize \geq 6$ then
        if $FillOrder(orderSize - 6)$ then
            return true
    if $orderSize \geq$ ____ then
        if $FillOrder($ ____ $)$ then
            return true
    if $orderSize \geq$ ____ then
        if $FillOrder($ ____ $)$ then
            return true
    return false
```

Question 4:

Show the tree of recursive calls made by the $FillOrder(orderSize)$ when initially called with $orderSize = 28$. If the same recursive call is made more than once in the tree, show the recursive subtree once, and then in subsequent calls just write `<PS>` for PREVIOUSLY SOLVED. The tree is wide, so draw it sideways. One each branch indicate the value returned either **true** or **false**. Be sure to also label the PREVIOUSLY SOLVED branches. (20 points)

? Question 5:
Based on your diagram, what is the total number of recursive calls made to `fillOrder` when the initial call is with an order of size 28? (2 points)

In class, we have been studying dynamic programming, an algorithm design technique that can be used to improve the efficiency of an optimization problem when a recursive solution has overlapping subproblems. In other words, it is used when the recursive implementation ends up making the same recursive calls multiple times. As you saw in your diagram, this happens with the Chicken McNuggets problem, and it can be very inefficient (often exponential or worse).

Using dynamic programming, a subproblem is solved only once and the answer is saved. If the solution to the same subproblem is needed again, instead of recomputing it, the previously saved answer is used. This form of dynamic programming is called top-down dynamic programming.

In your GitHub repository, you will find a program in `DynamicProgramming.java`. We will first work with the Chicken McNuggets problem. There is a complete implementation of an exhaustive search in the `fillOrderDynProg` method. Notice that an additional argument has been added: an array `S[]`. You will use `S[]` to convert this exhaustive search solution into a top-down dynamic programming solution. Specifically, `S[i]` will store -1 if subproblem `fillOrderDynProg(i)` has not yet been computed; `S[i]` will store 0 if subproblem `fillOrderDynProg(i)` was previously computed and determined to be `false`; and `S[i]` will store 1 if subproblem `fillOrderDynProg(i)` was previously computed and determined to be `true`. Initially all entries in `S[]` are set to -1 for you indicating that no subproblems have been solved.

? Question 6:
Modify the implementation of `fillOrderDynProg` so that it uses `S[]` as described above. If a subproblem has already been solved (so its solution is in `S[]`), then it should just return the answer stored in `S[]`. Otherwise, it will need to compute the answer to the subproblem by making recursive calls; in this case, be sure to store the answer in `S[]` after computing it! Demonstrate your modified implementation by passing the `testFillOrder` tests. (10 points)

? Question 7:
With your dynamic programming solution, what is the total number of non-trivial recursive calls made to `fillOrderDynProg(i)` when the initial call is with an order of size 28? A non-trivial recursive call is one in which the answer is not in `S[i]` and so it has to be computed. (6 points)

? Question 8:
Consider `fillOrderDynProg(100)`. Is it possible that $>$ 100 non-trivial recursive calls are made to `fillOrderDynProg`? Why or why not? (4 points)

? Question 9:
Based on the number of non-trivial recursive calls made, what is the worst case running time of `fillOrderDynProg(n)` expressed using order notation. (4 points)

For the remainder of this lab, you will be working with the Coin Row Problem. In this problem there is a row of n coins whose values are positive integers, not necessarily distinct. The goal is to pick up the maximum amount of money subject to the constraint that no two coins adjacent in the initial row can be picked up.

? Question 10:
Consider the row of coin values below. What is the maximum amount of money you can pick up? Circle the coins you would pick. (4 points)

1	**15**	**20**	**7**	**2**	**4**	**3**	**7**

Question 11:
Consider the row of coin values below. What is the maximum amount of money you can pick up? Circle the coins you would pick. (4 points)

2	**2**	**15**	**25**	**19**	**3**	**4**	**12**

To begin thinking about how you might solve this problem by solving smaller subproblems, suppose you can only see the first value in an eight item array, as shown below. Suppose further that you are told by someone who has seen the contents of the rest of the array, that 15 is the maximum amount of money you can pick up if you only take coins starting from the coin at index 1; also, 12 is the maximum you can pick up if you only take coins starting from the coin at index 2.

5	**?**	**?**	**?**	**?**	**?**	**?**	**?**

Question 12:
Can you determine from this information the maximum you can pick up using coins starting from the coin at index 0 in the array? If yes, what is the maximum? If no, explain why. (4 points)

Question 13:
In the class `DynamicProgramming`, complete the implementation of the method `maxPickUpCoins` so that it uses a top-down, **exhaustive search** approach to compute the maximum amount of money you can pick up. Demonstrate your method once it passes all of the tests in `testPickUpCoins`. (3 points)

? Question 14:

Show the tree of recursive calls made by `maxPickUpCoins(A, 0)` when called with the array `A = {12, 20, 10, 17, 16}`. If the same recursive call is made more than once in the tree, show the recursive subtree once, and then in subsequent calls just write ALREADY SOLVED. The tree is wide, so draw it sideways. On each branch indicate the value returned from that call, including branches that are marked ALREADY SOLVED. (20 points)

Question 15:

Complete the implementation of `maxPickUpCoinsDynProg(A, first, S)` that takes a third parameter, an array `S[]` that you should use to store solutions to subproblems that have been solved. If a subproblem is already solved, then just return the answer stored in `S[]` rather than resolving it. Initially all entries in `S[]` are initialized to -1 for you, indicating that none have been solved. Demonstrate your method once it passes all of the tests in `testPickUpCoinsDynProg`. (7 points)