Student Information

Name: _____ Student ID: _____
Due Date: 27 Nov 11:59pm.
Submit answers on eDimension in pdf format. Submission without student information will **NOT** be marked! Any questions regarding the homework can be directed to the TA through email (contact information on eDimension).

Week 11

1. In comparison to other methods that do not take advantage of overlapping subproblems, dynamic programming takes far less time in finding a solution. [True/False]. Explain why. ***Only half of the full marks will be awarded if answer is correct without explanation.***

   Solution: True. Other methods that do not take advantage of overlapping subproblems compute the solution of the subproblem several times, wasting time on recalculation. Dynamic programming solves the subproblems only once and thus save time on recalculation.

2. What are the changes in time and space complexities when a top-down approach of dynamic programming is applied to a problem?

   A. Time and space complexities decrease.
   B. Time and space complexities increase.
   C. Time complexity increases and space complexity decreases.
   D. Time complexity decreases and space complexity increases.

   Solution: D. Top-down approach uses the memoization technique which stores the previously calculated values.

3. Modify the rod-cutting problem in Figure 1 to include for each cut, a fixed cost c in addition to the price p_i for each rod. The revenue is now the sum of the prices of the pieces minus the costs of making cuts. Show the modified dynamic programming algorithm.
   Hint*: Modify line 4, 5 and 6.*

   Solution: Modified algorithm is in Figure 2. In line 6, the modification reflects the fixed cost c of making the cut. We also have to handle the case in which we make no cuts (when i equals to j); the total revenue in this case is simply $p[j]$. Thus, line 5 is modified to run from i to $j-1$. The assignment $q = p[j]$ takes care of the case of no cuts. Without these modifications, we would be deducting c from the total revenue even in the case where no cuts are made.

```
BOTTOM-UP-CUT-ROD(p, n)
1  let r[0..n] be a new array
2  r[0] = 0
3  for j = 1 to n
4      q = -∞
5      for i = 1 to j
6          q = max(q, p[i] + r[j - i])
7      r[j] = q
8  return r[n]
```

Figure 1: Rod-cutting algorithm

```
MODIFIED-CUT-ROD(p, n, c)
let r[0..n] be a new array
r[0] = 0
for j = 1 to n
    q = p[j]
    for i = 1 to j - 1
        q = max(q, p[i] + r[j - i] - c)
    r[j] = q
return r[n]
```

Figure 2: q3 ans

4. Dynamic programming algorithms are usually based on recursive equations. Why don't dynamic programming algorithms simply use recursion to implement those equations directly? Briefly explain.

   Solution: Computing the equations directly using recursion results in the same thing being computed many times, and causes the algorithm to be slow. Typically, the cost of the obvious recursive algorithm is the same as the cost of the algorithm that tries all solutions.

5. Let P(n) be the number of n-length binary strings that do not have any three consecutive ones (i.e. they do not have "111" as a substring). For example:

   P(1) = 2: 0, 1,
   P(2) = 4: 00, 01, 10, 11
   P(3) = 7: 000, 001, 010, 011, 100, 101, 110
   P(4) = 13: 0000, 0001, 0010, 0011, 0100, 0101, 0110, 1000, 1001, 1010, 1011, 1100, 1101
   ***Hint**: Each binary string has either one of the following 3 properties: a) Last bit is 0, b) Last two bits are 01, c) Last three bits are 011.*

(i) Write down the recursive formula to compute P(n).

(ii) Suppose we implement a dynamic programming algorithm using the recursive formula in (i), what is the running time of the dynamic programming algorithm?

Solution:
(i) Let $n \geq 4$. The number of strings with property a) is P(n-1), the number of strings with property b) is P(n-2), and the number of strings with property c) is P(n-3). Thus the recursive formula is
P(n) = P(n-1) + P(n-2) + P(n-3)

(ii) The running time is O(n). Refer to the code in Figure 3.

```
1: procedure TRIBUNACCI(n)
2:     p[1] = 2, p[2] = 4, p[3] = 7
3:     for i = 4 to n do
4:         p[i] = p[i − 1] + p[i − 2] + p[i − 3]
5:     return p[n]
```

Figure 3: q5 code