1. Consider a modification of the rod-cutting problem in which, in addition to a price p_i for each rod, each cut incurs a fixed cost of c. The revenue associated with a solution is now the sum of prices of the peices minus the cost of making the cut.

   (a) Give a dynamic-programming algorithm to solve this modified problem, including the mathematical expression for the maximum revenue and the pseudocode.

   (b) Show the maximum revenue r_j and the optimal size s_j of the first piece to cut off, when $c = 1$ and

length i	1	2	3	4	5	6	7	8	9	10
price p_i	1	4	5	9	10	12	15	18	19	20

Solution: (a) With each cut cost c, the optimal revenue r_n can be characterized as

$$r_n = \max\{p_n, \max_{1 \le i < n}\{p_i + r_{n-i} - c\}\}$$

The pseudo-code of the modified rod cutting problem is

```
Modified-Cut-Rod(p,n,c)
let r[0...n] and s[0...n] be new arrays
r[0] = 0
s[0] = 0
for j = 1 to n
    q = p[j]
    s[j] = j
    for i = 1 to j-1
        if q < p[i]+r[j-i]-c
           q = p[i]+r[j-i]-c
           s[j] = i
        end if
    end for
    r[j] = q
end for
return r and s
```

The major modification required is in the body of the inner for-loop. This change reflects the fixed cost of making the cut, which is deducted from the revenue. We also have to handle the case in which we make no cuts, namely, when `i` = `j`. The total revenue in this case is simply `p[j]`. For this, we modify the inner for-loop to run from `1` to `j-1` instead of to `j`. The assignment `q = p[j]` takes care of the case of no cuts.

(b) The r and s values are

length i	1	2	3	4	5	6	7	8	9	10
price p_i	1	4	5	9	10	12	15	18	19	20
Max-rev. r_i	1	4	5	9	10	12	15	18	19	21
first-cut s_i	1	2	3	4	5	6	7	8	9	2

2. Find an optimal parenthesization of a matrix-chain product $A_1A_2A_3A_4A_5A_6$, whose sequence of dimensions is $p = (5, 6, 3, 7, 5, 3, 4)$.

   (a) Compute the dynamic programming m-table and s-table.

   (b) Show the optimal parenthesization.

   Solution: (a) By the dynamic programming, the m-table and s-table are

```
M =                                                S =
       0     90    195    270    285    336            0     1     2     2     2     2
       0      0    126    195    204    258            0     0     2     2     2     2
       0      0      0    105    150    186            0     0     0     3     4     5
       0      0      0      0    105    189            0     0     0     0     4     5
       0      0      0      0      0     60            0     0     0     0     0     5
       0      0      0      0      0      0            0     0     0     0     0     0
```

   (b) By the s-table, an optimal parenthesization is

$$((A_1A_2)(((A_3A_4)A_5)A_6))$$

   and by the m-table, the corresponding (least) operations is 336.

3. For the sequences $X = \langle B, C, A, A, B, A\rangle$ and $Y = \langle A, B, A, C, B\rangle$,

   (a) Follow the pseudocode LCS-LENGTH to fill in the dynamic programming c- and b-tables for finding the longest common subsequence (LCS) of X and Y.

   (b) Follow the pseudocodes PRINT-LCS, list the LCS.

   Solution: (a) Following the pseudocodes LCS-LENGTH, we obtain the $c - b$ table:

		0	1	2	3	4	5
		y_j	A	B	A	C	B
0	x_j	0	0	0	0	0	0
1	B	0	$\uparrow 0$	$\nwarrow 1$	$\leftarrow 1$	$\leftarrow 1$	$\nwarrow 1$
2	C	0	$\uparrow 0$	$\uparrow 1$	$\uparrow 1$	$\nwarrow 2$	$\leftarrow 2$
3	A	0	$\nwarrow 1$	$\uparrow 1$	$\nwarrow 2$	$\uparrow 2$	$\uparrow 2$
4	A	0	$\nwarrow 1$	$\uparrow 1$	$\nwarrow 2$	$\uparrow 2$	$\uparrow 2$
5	B	0	$\uparrow 1$	$\nwarrow 2$	$\uparrow 2$	$\uparrow 2$	$\nwarrow 3$
6	A	0	$\nwarrow 1$	$\uparrow 2$	$\nwarrow 3$	$\leftarrow 3$	$\uparrow 3$

   The length of the LCS is 3.

   (b) By following the pseudocode PRINT-LCS, an LCS is BCB (why not the LCS ABA or others?)

4. Two character strings may have many common substrings. Substrings are required to be contiguous in the original string. For example, *photograph* and *tomography* have several common substrings of length one (i.e., single letters), and common substrings *ph*, *to*, and *ograph* (as well as all the substrings of *ograph*). The maximum common substring (MCS) length is 6.

   Let $X = x_1 x_2 \cdots x_m$ and $Y = y_1 y_2 \cdots y_n$ be two character strings.

   (a) Give a dynamic programming algorithm to find the MCS length for X and Y.

   (b) Analyze the worst-case running time and space requirements of your algorithm as functions of n and m.

   (c) Demonstrate your dynamic programming algorithm for finding the MCS length of character strings *cabccb* and *babcba* by constructing the dynamic programming tables.

Solution: (a) We use a table $D[0..m, 0..n]$, where $d[i,j]$ is the length of an MCS ending at x_i and y_j. The first row and column are initialized to be zero. Then the remaining entries (by the optimal substructure of MCS) are computed by the following recursive formula: for $i = 1, 2, \ldots, m$ and $j = 1, 2, \ldots, n$:

$$d[i,j] = \begin{cases} d[i-1, j-1] + 1 & \text{if } x_i = y_j \\ 0 & \text{otherwise} \end{cases}$$

The table entries can be computed row-by-row. The answer to the problem is the largest entry of D.[1] Based on the above formula, we can write down a pseudocode for computing the MCS, similar to the LCS pseudocode *(omit)*.

(b) The running time of this procedure is $\Theta(mn)$, since each table entry takes $O(1)$ to compute.

(c) Here is the d-table:

		b	a	b	c	b	a
	0	0	0	0	0	0	0
c	0	0	0	0	1	0	0
a	0	0	1	0	0	0	1
b	0	1	0	2	0	1	0
c	0	0	0	0	3	0	0
c	0	0	0	0	1	0	0
b	0	1	0	0	0	2	0

By the d-table, the length of MCS of the character strings *cabccb* and *babcba* is 3. The MCS is *abc*.

[1] An alternative solution uses a table $M[1..m, 1..n]$ where

$$m[i,j] = \begin{cases} 1 & \text{if } x_i = y_j \\ 0 & \text{otherwise} \end{cases}$$

After filling the table, the algorithm scans all diagonals for the longest string of contiguous 1's. Each such string corresponds to a common substring. This method also takes $\Theta(mn)$.

For the first solution, only two rows need actually be stored. It is easy to keep track of the largest value computed so far while filling the table. The second solution uses more space, it requires to store the the whole $m \times n$ table.

5. This problem continues Problem 6 of Homework #4 to solve the following 0-1 knapsack problem by using dynamic programming:

   Given six items $\{(v_i, w_i)\}$ for $i = 1, 2, \ldots, 6$ as follows:

i	v_i	w_i
1	40	100
2	35	50
3	18	45
4	4	20
5	10	10
6	2	5

   and the total weight $W = 100$, where v_i and w_i are the value and weight of item i, respectively. Compute the solution by dynamic programming and comment your findings.

   You need to outline your DP algorithm, not just given the answer. Since W is pretty big, you need to write a short program to do the calculation and submit a hardcopy of your program.

   Solution: A short code in C++ to implement the the dynamic programming of the 0-1 knapsack problem is shown in the next page. The last column of the following table shows the solution given by the dynamic programming.

			Greedy by			**optimal solution**
i	v_i	w_i	value	weight	v_i/w_i	**by dynamic prog.**
1	40	100	1	0	0	0
2	35	50	0	0	1	**1**
3	18	45	0	1	0	**1**
4	4	20	0	1	1	0
5	10	10	0	1	1	0
6	2	5	0	1	1	**1**
Total value			40	34	51	**55**
Total weight			100	80	85	**100**

   All three greedy approaches generate feasible solutions, but none of them generate the optimal solution. On the other hand, the solution given by the dynamic programming is optimal.

```
//  0-1 Knapack problem
//  Created by Wei-Chih Chen on 2/10/16.
#include <iostream>
using namespace std;
int main(){
    int n = 6;
    int W = 100;
    int v[7] = {0,40,35,18,4,10,2};
    int w[7] = {0,100,50,45,20,10,5};
    int c[n+1][W+1];
    //initial base case
    for(int i=0;i<=n;i++){
        c[i][0] = 0;
    }
    for(int i=1;i<=W;i++){
        c[0][i] = 0;
    }
    //generate c-table
    for(int i=1;i<=n;i++){
        for(int weight=1;weight<=W;weight++){
            if(w[i]<=weight){
                if(v[i]+c[i-1][weight-w[i]] > c[i-1][weight])
                    c[i][weight] = v[i] + c[i-1][weight-w[i]];
                else
                    c[i][weight] = c[i-1][weight];
            }else{
                c[i][weight] = c[i-1][weight];
            }
        }
    }
    //print out solution
    int w0 = W;
    for(int i = n; i>0; i--){
        if(c[i][w0] !=c[i-1][w0]){
            w0 = w0 - w[i];
            cout << i << "\n";
        }
    }
    //print out c table
    /*
    for(int i =0;i<=n;i++){
        for(int weight=0;weight<=W;weight++){
            cout<< c[i][weight] << " ";
        }
        cout << "\n";
    }
    */
    return 0;
}
```