Xiaoying Li
CS 325 -Winter 2020
Homework #3

- **Problem 1**

Common:

Dynamic programming and divide-and-conquer both solve problems by combining solutions to subproblems.

Difference:

Unlike divide-and-conquer, dynamic programming's subproblems are not unique. Dynamic programming's subproblems may share sub subproblems. However, solution to one subproblem may not affect the solutions to other subproblems of the same problem.

- **Problem 2**

a.

Explanation:

Number the rows and columns of the chessboard with 1 to 8 from left to right and top to bottom. Assume the chess rock is initially located in the square (1, 1). Let N(i, j) be the number of shortest paths the chess rock can move from square (1, 1) to square (i, j) ($1 \leq i, j \leq 8$). A chess rook can move horizontally or vertically to any square in the same row or the same column, so N(i, 1)=N(1, j)=1 and any shortest path to square (i, j) must pass through its adjacent square (i, j-1) or square (j, i-1). So, the recurrence for the number of shortest paths the chess rock can move from square (1, 1) to square (i, j) can be stated as:

$$N(i,j) = N(i,j-1) + N(j,i-1)$$
$$N(i,1) = N(1,j) = 1, 1 \leq i,j \leq 8$$

And because the chessboard is symmetric, so N(i, j)=N(j, i). Thus, we only need to calculate P(i, j) above or below the main diagonal.

Pseudocode of dynamic programming algorithm:

Memorized DP Algorithm:

```
symmetric square = { }; //store N(i, j) of squares having symmetric squares.
N (i, j) {
    if (square (j, i) is in symmetric square { } ) {
        return symmetric square [(j, i)];
    }
    if (i=1 or j=1) {
        N (i, j) = 1;
    }
```

```
        else {
            N (i, j) = N (i, j-1) + N (i-1, j);
        }
        symmetric square [(i, j)] = N (i, j);
        return N (i, j);
    }
```

Bottom-up DP Algorithm:

```
    N { }
    for 1 <= i <= 8
        N [(i, 1)] = 1;
    for 1 <= j <= 8
        N [(1, j)] = 1;
    for 2 <= i <= 8 {
        for 2 <= j <= 8 {
            N [(i, j)] = N [(i, j-1)] + N [(i-1, j)];
    }
    return N [(8, 8)];
```

Sub-problems results and final result:

Using the recurrence with overlapping subproblems, the values of each N(i, j) can be calculated, and the results are shown below:

1	1	1	1	1	1	1	1
1	2	3	4	5	6	7	8
1	3	6	10	15	21	28	36
1	4	10	20	35	56	84	120
1	5	15	35	70	126	210	330
1	6	21	56	126	252	462	792
1	7	28	84	210	462	924	1716
1	8	36	120	330	792	1716	3432

Therefore, the final result of the number of shortest paths by which a rook can move from one corner of a chessboard to the diagonally opposite corner is 3432.

b.

Any shortest paths by which a rook can move from one corner of a chessboard to the diagonally opposite corner are 14 consecutive moves to adjacent squares composed of 7 moves being up or down and the other 7 being left or right. So, the total number of such paths is the number of different ways to choose 7 moves among 14 moves to go up or down (and the other 7 being left or right), which is:

$$C(14,7) = \frac{14!}{7! \cdot 7!} = 3432$$

Therefore, the final result of the number of shortest paths by which a rook can move from one corner of a chessboard to the diagonally opposite corner is 3432.

- **Problem 3**

Explanation:

If the given matrix B only have one row or one column, then any cell whose element is zero is B's largest square submatrix whose elements are all zeros, and its size=1. [Base Case]

Create an auxiliary matrix A of the same size m*n as the given matrix B. Fill every cell of the auxiliary matrix A[i][j] with the size of largest square submatrix who includes B[i][j] as its rightmost and bottommost and are all zeros. [Bottom-up Dynamic Programming Algorithm]

After filling all auxiliary matrix A's cells, scan it and find the maximum value, the square of the maximum value is exactly the size of the given matrix B's largest square submatrix whose elements are all zeros. [Final Result]

Below is an example:

1	1	1	0	→	0	0	0	1
1	1	0	0	→	0	0	1	1
1	0	0	0	→	0	1	1	2
0	0	0	0	→	1	1	2	2
1	0	0	0	→	0	1	2	3
1	0	1	0	→	0	1	0	1

Pseudocode:

Create an auxiliary matrix A[m][n] of B[m][n];
For first row and first column of A[m][n],
    if B[1][j]=0, then A[1][j]=1;
    if B[i][1]=1, then A[i][1]=0.
For other cells of A[m][n],
    if B[i][j]=0, then A[i][j]=minimum(B[i-1][j], B[i][j-1], B[i+1][j+1])+1;

if B[i][j]=1, then A[i][j]=0.

Scan and find the maximum value of A[m][n],

print the maximum value m*m and its index A[x][y];

print the rightmost and bottommost index of B's largest square submatrix B[x][y];

print the leftmost and topmost index of B's largest square submatrix B[x-m+1][y-m+1].

print B's largest square submatrix.

Time efficiency:

O(m*n), m is the number of rows and n is the number of columns of the given matrix.

- **Problem 4**

a.

Let V[i, j] be the optimal value to this instance of the knapsack problem, then according to "Lecture Note – 0-1 Knapsack Problem", the bottom-up dynamic programming algorithm of the knapsack problem can be stated as below:

$Base\ case: V[i,0] = V[0,i] = 0, i > 0, j > 0$

$Recurrence: V[i,j] = \begin{cases} \max\{V[i-1,j], V_i + V[i-1, j-w_i]\}, if\ j \geq w_i \\ V[i-1,j], if\ j < w_i \end{cases}$

Apply this algorithm to the instance of the knapsack problem:

I \ J	0	1	2	3	4	5	6
0 (w, v)	0	0	0	0	0	0	0
1 (3, 25)	0	0	0	25	25	25	25
2 (2, 20)	0	0	20	25	25	45	45
3 (1, 15)	0	15	20	35	40	45	60
4 (4, 40)	0	15	20	35	40	55	60
5 (5, 50)	0	15	20	35	40	55	65

The max possible value that can be carried in the knapsack is V[5, 6]=65.

b.

From part (a) we know that the max possible value that can be carried in the knapsack is V[5, 6]=65. For V[5, 6], $j = 6, w_i = w_5 = 5$, so $j \geq w_i$. Thus,

$$\begin{aligned} V[5,6] &= \max\{V[i-1,j], V_i + V[i-1, j-w_i]\} \\ &= \max\{V[5-1,6], V_5 + V[5-1, 6-w_5]\} \\ &= \max\{V[4,6], 50 + V[4,1]\} \\ &= \max\{60, 50+15\} \\ &= 65 \end{aligned}$$

Trace backward the computation of V[5, 6], $V[5,6] = V_5 + V[4,1]$. And for V[4, 1], $j = 1, w_i = w_4 = 4$, so $j < w_i$. Thus V[4, 1]=V[4-1, 1]=V[3, 1]. For V[3, 1], $j = 1, w_i = w_3 = 1$, so $j \geq w_i$. Thus,

$$V[3,1] = \max\{V[i-1,j], V_i + V[i-1, j-w_i]\}$$
$$= \max\{V[3-1,1], V_3 + V[3-1, 1-w_3]\}$$
$$= \max\{V[2,1], 15 + V[2,0]\}$$
$$= \max\{0, 15+0\}$$
$$= 15$$

So, $V[3,1] = V_3$.

Hence, $V[5,6] = V_5 + V[4,1] = V_5 + V[3,1] = V_5 + V_3$. Therefore, the instance of part (a) only have one optimal subset, which is {item 3, item 5}.

c.

According to the computation of every V[i, j] in part (a) and the procedure of finding out the optimal subset in part (b), the only situation can cause the knapsack problem's instance has more than one optimal subset is $when\ j \geq w_i, V[i-1,j] = V_i + V[i-1, j-w_i]$.

Therefore, $when\ j \geq w_i, if\ V[i-1,j] = V_i + V[i-1, j-w_i]$, then there is more than one optimal subset for the knapsack problem's instance.

e.

Time efficiency:

The outer loop runs at most n times; the inner loop runs at most W times. So, the time efficiency for the bottom-up dynamic programming algorithm is in $\theta(nW)$.

Space efficiency:

This bottom-up dynamic programming algorithm needs to allocate an extra matrix to store the value of every subproblem. And the size of the matrix is (n+1)*(W+1)=nW+n+W+1, so the space efficiency for this algorithm is also in $\theta(nW)$.

Time needed to find the composition of an optimal subset from a filled dynamic programming table:

To do this, first we need to find the optimal value in the filled table. We just need to use the index to access the corresponding element, so its time complexity is O(1).

Then we need to use the optimal value to trace backward the table to find the composition of an optimal subset, so its time complexity is O(n)

Therefore, the time needed to find the composition of an optimal subset from a filled dynamic programming table is in O(1)+O(n)=O(n).