Problem #1 - Exercise 8.1-2 (page 196)

Prove that Counting Sort is Stable

Initial Call to function Counting Sort: CountingSort(Unsorted_Array, Sorted_Array, max(Unsorted_Array))

	CountingSort(A, B, k)	
1	n = A.length	
2	Let C[0 to k] be new array of int	
3	**For**(j = 0 **to** k)	
4	C[j] = 0	**//---- Initialize the array C to all 0's**
5	**For**(j = 0 **to** n)	
6	C[A[j]] = C[A[j]] +1	**//---- Push element A[j] onto the stack at C[A[j]]**
7	**For**(j = 1 **to** k)	
8	C[j] = C[j] + C[j -1]	**//---- Place all previous, lower value stacks under the stack at C[j]**
9	**For**(j = n **downto** 1)	
10	B[C[A[j]]] = A[j]	**//----- Since stacks are FILO, populate B with the last equivalent value near to the end of the array. This is determined by knowing the place in the relative stack from lines #7-8**
11	C[A[j]] = C[A[j]] -1	**//----- Pop the last element in stack C[A[j]] off the stack**

Stable sorting algorithms maintain the relative order of equivalent elements in the unsorted source array (e.g. *A*).

Counting sort achieves stability by leveraging the characteristics of the stack data structure. A stack maintains order by always placing new elements on top of all previous elements (i.e. as opposed to inserting the elements at the beginning or in the middle of the list of previous elements).

In the pseudcode above, array *C* is essentially *k+1* stacks. Each stack stores the elements that have a specific integer value between *0* and *k*.

Step#1: Lines 5 and 6 are where the individual stacks (i.e. array C) are populated. Starting with the first element in *A*, elements are pushed onto the *k+1* stacks based on the value of that element. If an element A[*i*] (where $1 \le i \le n$) has the same value as an element A[*j*] (where $1 \le j \le i-1$), then element A[*i*] is pushed onto the stack C[A[*i*]] on top of A[*j*].

Step#2: Since all stacks C[i] (where $0 \le i \le k$) each individually maintain the relative order of equivalent elements, then stacking all lower value stacks C[j] (where *j* is 0 to *i*-1) below stack C[*i*] still maintains stability since now stack C[i], is now on top of the stacks C[0] → C[1] → C[2] → … → C[i-1]. As such, the stacks are individually (i.e. for equivalent elements) and collectively (i.e. for all lower value elements) ordered. This is done in lines 7 and 8 above.

Step#3: All stacks have the first in last out (FILO) property. To maintain stability, the FILO property must be used properly in that the element on the top of the stack must be placed later in the sorted array than those elements below it on the stack.

That is why in lines #9 and #10 above, the unsorted source array *A* is read from the end of the array to the beginning of the array. By doing so, the sorted array B is populated from the last element to the first element by reading merged stacks in array C (from lines #7-8). In line 11, the top of the stack for element value A[i] is

popped off before any values below it on the stack (e.g. A[*j*]). This ensures that the order of equivalent values in the original array A is maintained.

Problem #2

Show how to sort integers in the range 1 to n^d in Θ(dn).

Lemma 8.3 of the textbook says that given a set of n d-digit numbers where each digit can take up to k possible values, Radix Sort can sort this set of numbers in Θ(d*(n+k)) time if the stable sort algorithm it uses takes Θ(n+k) time.

Step #1: The n integers in this question are by definition in base-10. Convert the integers from base-10 to base-n. This can be done in Θ(dn) time. To convert, each number in the range 1 to n^d to base-n takes at most d+1 divisions; hence, to convert all n digits takes Θ((d+1)*n), which is Θ(dn).

Step #2: Sort the n numbers now in base-n using radix sort. Each number n has a maximum of d non-zero digits, and each digit has a maximum of n possible values. As such based off lemma 8.3 mentioned above, the running time of this step is:

$$\theta(Step_2) = \theta\big(d(n+k)\big)$$

$$\theta(Step_2) = \theta\big(d(n+n)\big)$$

$$\theta(Step_2) = \theta(d \cdot 2n)$$

$$\theta(Step_2) = \theta(dn)$$

The total running time of this algorithm is then the combined running time of the two steps:

$$\theta(Total) = \theta(Step_1) + \theta(Step_2)$$

$$\theta(Total) = \theta(dn) + \theta(dn)$$

$$\theta(Total) = \theta(dn)$$

If d is held constant, then for n much larger than d, this simplifies to:

$$\theta(Total) = \theta(n)$$

Problem #3 - Exercise 15.1-3 (page 370)

Modify the cut rod algorithm to handle the case where cutting the rod has a cost. Array "prices" is an array of length 0 to n and is the value contains the values for a rod of length *i*, where *i* is the index of the array.

Initial Call: `int max_revenue = Q3_Cut_Rod_With_Cost(Q3_n, prices, c, out s);`

```
static int Q3_Cut_Rod_With_Cost(int n, int[] prices, int cost_per_cut, out int[] s)
{
    int[] r = new int[n+1];
    s = new int[n+1]; //----- S is a two dimensional array containing the cuts for indexes 1 to n
    int i, j;
    int r_temp;

    //---- Iterate through the possibilities
    for (i = 1; i < n+1; i++)
    {
        r[i] = prices[i];
        s[i] = i;
        for (j = i-1; j >= i/2; j--)//---- Use of i/2 is explained in note below
        {
            r_temp = r[j] + r[i - j] - cost_per_cut;
            if (r_temp > r[i])
            {
                r[i] = r_temp;
                s[i] = i-j;
            }
        }
    }

    return r[n];
}

static void Q3_Print_Cut_Rod_Pieces(int n, int[] s)
{
    string str = s[n].ToString() ;
    n -= s[n];
    while (n > 0)
    {
        str += ", " + s[n].ToString();
        n -= s[n];
    }
    Console.WriteLine("The rod pieces are length: " + str);
}
```

Note #1: Before I begin, it is important I note that the inner for loop in function "Q3_Cut_Rod_With_Cost" is different than the textbook so it may ***look*** wrong. However, the textbook uses a simplified version of the algorithm to get to $O(n^2)$. My algorithm above is:

$$\theta\left(\frac{n^2}{2}\right) \in \theta(n^2)$$

It is not asymptotically faster; it just has lower constant factors. This modification is possible because the algorithm included in the book essentially calculates r(j) and r(n-j) twice. My algorithm eliminates this duplication. The question

did not specifically ask I go to this step, but I felt it would be possible and fun to do so. As such, I figured out how to handle it.

Note #2: Second, C/C++/C# are all base 0; as such, I had to modify the initialization of the array to be size "n+1" to enable my code to work on an array of size 1 to *n*, as is done in the textbook.

Algorithm summary:

Outer Loop: For each length *i* (where $1 \le i \le n$), the algorithm calculates the maximum revenue for that length. The algorithm iterates through all possible rod lengths by first handling the special case where the rod of length *i* is not cut. In that case, the cost_per_cut (i.e. *c*) is ignored as no cuts are required.

Inner loop: The algorithm iterates through the different possible rod lengths *j* (where *j* is defined as *Floor*(i/2)[1] $\le j \le i$ - 1[2]), and calculates the revenue if the rod is cut into pieces of length *j* and *i-j*. The algorithm sums the revenue r_j (for a rod of length *j*) and r_{i-j} (for a rod of length *i-j*) and subtracts the cost of that associated cut. If the revenue of this new set of rods is greater than the maximum value found so far, this new maximum revenue is stored as well as the length of the minimum cut (i.e. *i-j*).

Reconstruction algorithm: Given a rod of length *n*, the function reads in an array *s* which contains the optimal cut length given a rod of size *i* (where $1 \le i \le n$). The function starts with a rod of length *n* and traverses through the array *s* until all cuts are made (i.e. n=0). This function is similar to the print function in the textbook.

[1] Floor is not written in algorithm code as it is inherent in C/C++/C# integer division.
[2] Loop only needs to start at *i*-1 because the case of *i* is handled outside the loop as a special case since it requires no cuts.

Problem #4 - Exercise 15.4-1, page 396

X = (1, 0, 0, 1, 0, 1, 0, 1)

Y = (0, 1, 0, 1, 1, 0, 1, 1, 0)

			Y									
			0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
			y_j	**0**	**1**	**0**	**1**	**1**	**0**	**1**	**1**	**0**
	0	x_i	0	0	0	0	0	0	0	0	0	0
	1	**1**	0	↑ 0	↖ 1	← 1	↖ 1	↖ 1	← 1	↖ 1	↖ 1	← 1
	2	**0**	0	↖ 1	↑ 1	↖ 2	← 2	← 2	↖ 2	← 2	← 2	↖ 2
	3	**0**	0	↖ 1	↑ 1	↖ 2	↑ 2	↑ 2	↖ 3	← 3	← 3	↖ 3
X	**4**	**1**	0	↑ 1	↖ 2	↑ 2	↖ 3	↖ 3	↑ 3	↖ 4	↖ 4	← 4
	5	**0**	0	↖ 1	↑ 2	↖ 3	↑ 3	↑ 3	↖ 4	↑ 4	↑ 4	↖ 5
	6	**1**	0	↑ 1	↖ 2	↑ 3	↖ 4	↖ 4	↑ 4	↖ 5	↖ 5	↑ 5
	7	**0**	0	↖ 1	↑ 2	↖ 3	↑ 4	↑ 4	↖ 5	↑ 5	↑ 5	↖ 6
	8	**1**	0	↑ 1	↖ 2	↑ 3	↖ 4	↖ 5	↑ 5	↖ 6	↖ 6	↑ 6

$$Z = (y_2, y_3, y_6, y_7, y_8, y_9) = (x_1, x_2, x_3, x_4, x_6, x_7) = (1, 0, 0, 1, 1, 0)$$

Source Code for Verification - Full Program is in Appendix #1

```
static void Q4_Longest_Common_Subsequence(int[] X, int[] Y, out LCS_Cell[,] LCS_Matrix){
    int m = X.Length-1;
    int n = Y.Length-1;
    int i, j;
    LCS_Cell temp_LCS_Cell;

    //---- Initialize array referred to as b and c
    LCS_Matrix = new LCS_Cell[m + 1, n + 1];

    //----- Initialize the arrays
    temp_LCS_Cell.dir = LCS_Dir.End;
    temp_LCS_Cell.lcs_len = 0;
    for (i = 0; i <= m; i++) LCS_Matrix[i, 0] = temp_LCS_Cell;
    for (j = 0; j <= n; j++) LCS_Matrix[0, j] = temp_LCS_Cell;

    //----- Iterate through all cells in the array and generate the matrix values
    for (i = 1; i <= m; i++)
    {
        for (j = 1; j <= n; j++)
        {
            //---- X[i] and Y[i] are the same so mark as part of a sequence
            if (X[i] == Y[j])
            {
                temp_LCS_Cell.dir = LCS_Dir.Diagonal;
```

```
                temp_LCS_Cell.lcs_len = LCS_Matrix[i-1,j-1].lcs_len + 1;
            }

            //---- X[i] and Y[j] not in the same sequence so point to longest subsequence
            else if(LCS_Matrix[i-1,j].lcs_len >= LCS_Matrix[i,j-1].lcs_len){
                temp_LCS_Cell.dir = LCS_Dir.Up;
                temp_LCS_Cell.lcs_len = LCS_Matrix[i-1,j].lcs_len;
            }
            else{
                temp_LCS_Cell.dir = LCS_Dir.Left;
                temp_LCS_Cell.lcs_len = LCS_Matrix[i, j-1].lcs_len;
            }
            LCS_Matrix[i, j] = temp_LCS_Cell; //--- Store data structure into matrix
        }
    }
}

static void Q4_Print_Subsequence(int[] X, LCS_Cell[,] LCS_Matrix)
{
    int i = LCS_Matrix.GetLength(0)-1;
    int j = LCS_Matrix.GetLength(1)-1;
    string print_str = "";

    while (LCS_Matrix[i, j].dir != LCS_Dir.End)
    {
        if (LCS_Matrix[i, j].dir == LCS_Dir.Diagonal)
        {
            //---- Generate sequence
            if (print_str == "") print_str = X[i].ToString();
            else print_str = X[i].ToString() + ", " + print_str;
            i--;
            j--;
        }
        else if (LCS_Matrix[i, j].dir == LCS_Dir.Left) j--;

        else                                           i--;
    }

    Console.WriteLine("A longest common subsequence is: {0}.", print_str);
}
```

Problem #5 - Exercise 15.4-2, page 396

C# Implementation of Reconstruct Algorithm – See Appendix for Full Source Code

```
static void Q5_Reconstruct_LCS(int[] X, int[] Y, LCS_Cell[,] LCS_Matrix)
{

    //---- Subtracting since C/C++/C# start at index 0 so have dummy index 0 increasing length by 1
    int i = X.Length - 1;
    int j = Y.Length - 1;
    string print_str = "";

    while (i > 0 && j > 0)
    {
        //----- In this case, the two values are equal, move diagonally
        if (X[i] == Y[j])
        {
            if (print_str == "") print_str = X[i].ToString();
            else print_str = X[i].ToString() + ", " + print_str;
            i--;
            j--;
        }
        //----- Elements do not match so take the path (up or left) with the longest common subsequence
        else if (LCS_Matrix[i - 1, j].lcs_len >= LCS_Matrix[i, j - 1].lcs_len) i--;
        else j--;
    }

    Console.WriteLine("A longest common subsequence is: {0}.", print_str);
}
```

Simplified Pseudo Code

X, Y are arrays (i.e. sequences) of integers. c is a two dimensional matrix of size X.length by Y.length.

```
void Q5_Reconstruct_LCS(X, Y, c)
{

    //---- Subtracting since C/C++/C# start at index 0 so have dummy index 0 increasing length by 1
    i = X.Length
    j = Y.Length
    output = ""

    while (i > 0 And j > 0)
    {
        //----- In this case, the two values are equal, move diagonally
        if (X[i] == Y[j])
        {
            if (output == "") output = X[i].ToString()
            else output = X[i] + ", " + output
            i = i - 1
            j = j - 1
        }
        //----- Elements do not match so take the path (up or left) with the longest common subsequence
        else if (c[i - 1, j] >= c[i, j - 1])
              i = i - 1
        else
              j = j - 1
    }
```

```
    Print output
}
```

Algorithm Runtime Analysis: The while loop defines the asymptotic runtime of this algorithm. It ends when either *i* or *j* equals zero. Initial values of *i* and *j*:

1. *i* = *m* = X.Length
2. *j* = *n* = Y.Length

Each time the loop is run, there are three possible case:

1. Only *i* is decremented
2. Only *j* is decremented
3. Both *i* and *j* are decremented

Worst case runtime is each time through the loop, only i or j is decremented (never both). Either *i* or *j* is decremented to 1 and the other is decremented to 0. The runtime in that case is defined by:

$$O(m + n - 1) \in O(m + n)$$

Algorithm Correctness Summary: Algorithm starts at cell (*m*, *n*) in matrix *c*; two iterators, *i* and *j*, are initialized to *m* and *n* respectively. With each iteration through the while loop, one of the four cases happens:

1. *i*=0 and/or *j*=0 ending the search through the matrix/subsequences.
2. X[i] and Y[j] are the same symbol. This symbol, X[i]/Y[j] is prepended onto the sequence string, and *i* and *j* are decremented.
3. X[i] and Y[j] are different, and the maximum (or an equivalent) length subsequence is found by decrementing the iterator *i* on sequence X so *i* is decremented.
4. X[i] and Y[j] are different, and the maximum length subsequence is found by decrementing the iterator *j* on sequence Y so *j* is decremented.

At the end, the sequence is printed.

Problem #6

Maximum Subarray Using Dynamic Programming

Input to this function is an array list_of_prices[1 to *n*] of values (e.g. prices of a stock as given in the book example). *n* is the number of elements in the array "list_of_prices".

C# Implementation of Maximum Subarray

```
static Tuple<int, int, int> Q6_Dynamic_Max_Subarray(int[] list_of_prices, int n)
{
    Tuple<int, int, int> Output_Results;
    int max_subarray = -1, max_subarray_start=-1, max_subarray_end =-1;
    PriceStruct[] MaxPrice, MinPrice;
    PriceStruct temp_price;
    int i;

    //----- Initialize arrays containing the maximum and minimum prices for the previous segments
    MaxPrice = new PriceStruct[n+1]; //--- MaxPrice is the maximum price from index i to index n
    MinPrice = new PriceStruct[n+1]; //--- MinPrice is the minimum price from index 1 to index i

    //----- For each day i = 1 to n, find the max price between that day and all later days
    temp_price.price = list_of_prices[n];
    temp_price.day = n;
    MaxPrice[n] = temp_price;
    for (i = n - 1; i >= 1; i--)
    {
        //----- Check if current price is higher than all previous prices
        if (list_of_prices[i+1] > MaxPrice[i + 1].price)
        {
            temp_price.price = list_of_prices[i+1];
            temp_price.day = i+1;
            MaxPrice[i] = temp_price;
        }
        else MaxPrice[i] = MaxPrice[i+1];
    }

    //----- For each day i = 1 to n-1, find the min price for all days before that day
    temp_price.price = list_of_prices[1];
    temp_price.day = 1;
    MinPrice[1] = temp_price;
    for (i = 2; i < n; i++)
    {
        //----- Check if current price is higher than all previous prices
        if (list_of_prices[i] < MinPrice[i-1].price)
        {
            temp_price.price = list_of_prices[i];
            temp_price.day = i;
            MinPrice[i] = temp_price;
        }
        else MinPrice[i] = MinPrice[i - 1];
    }


    //---- Iterate through the days to find the maximum profit made by selling on each day.
    max_subarray = int.MinValue;//---- Set to minimum value so always overwritten
    for (i = 1; i < n; i++)
    {
        if (MaxPrice[i].price - MinPrice[i].price > max_subarray)
        {
            max_subarray = MaxPrice[i].price - MinPrice[i].price;
```

```
            max_subarray_start = MinPrice[i].day;
            max_subarray_end = MaxPrice[i].day;
        }
    }


    //----- Return the results
    Output_Results = Tuple.Create<int, int, int>(max_subarray, max_subarray_start, max_subarray_end);
    return Output_Results;
}
```

Pseudocode Implementation of Maximum Subarray

```
Q6_Dynamic_Max_Subarray(int[] list_of_prices, int n)
{

    //----- Initialize arrays containing the maximum and minimum prices for the previous segments
    Let MaxPrice[1 to n] Be Array Int //--- MaxPrice is the maximum price from index i to index n
    Let MaxPrice_Day[1 to n] Be Array Int //--- MaxPrice is the day with max price from index i to n
    Let MinPrice [1 to n] Be Array Int //--- MinPrice is the minimum price from index 1 to index i
    Let MinPrice_Day[1 to n] Be Array Int //--- MinPrice is the day with min price from index 1 to i


    //----- For each day i = 1 to n, find the max price between that day and all later days
    MaxPrice_Day[n] = n;
    MaxPrice[n] = list_of_prices[n]
    for i = n - 1 downto 1
        //----- Check if current price is higher than all previous prices
        if (list_of_prices[i+1] > MaxPrice[i + 1])
            MaxPrice[i] = list_of_prices[i+1]
            MaxPrice_Day[i] = i+1

        else
            MaxPrice[i] = MaxPrice[i+1]
            MaxPrice_Day[i] = MaxPrice_Day[i+1]


    //----- For each day i = 1 to n-1, find the min price for all days before that day
    MinPrice_Day[i] = 1;
    MinPrice[1] = list_of_prices[1];
    for i = 2 to n - 1
        //----- Check if current price is higher than all previous prices
        if (list_of_prices[i] < MinPrice[i-1])
            MinPrice[i] = list_of_prices[i]
            MinPrice_Day[i] = i

        else
            MinPrice[i] = MinPrice [i-1]
            MinPrice_Day[i] = MinPrice_Day[i-1]


    //---- Iterate through the days to find the maximum profit made by selling on each day.
    max_subarray = -∞ //---- Set to minimum value so always overwritten
    for i = 1 to n - 1
        if (MaxPrice[i] - MinPrice[i] > max_subarray)
            max_subarray = MaxPrice[i] - MinPrice[i]
            max_subarray_start = MinPrice_Day[i]
            max_subarray_end = MaxPrice_Day[i]
        i = i + 1


    //----- Return the results
    return max_subarray, max_subarray_start, max_subarray_end
}
```

Algorithm Runtime Analysis: This algorithm has three for loops. The first and the third loop run *n*-1 times while the second loop runs *n*-2 times. The runtime of this algorithm is:

$$T(n) = O(n-1) + O(n-2) + O(n-1) + O(c)$$

$$T(n) = O(n) + O(n) + O(n) + O(1)$$

$$T(n) = O(\max\{n, n, n, 1\})$$

$$T(n) = O(\mathrm{n})$$

Correctness: The maximum subarray problem is built on an optimal substructure. Given an input array A and element A_i where $1 \le i \le n$, then the maximum subarray ($MaxSubarray_i$) starting on element *i* is defined as:

$$MaxSubarray_i = \max_{i+1 \le j \le n} \{A_j\} - A_i$$

In simple terms, if the maximum subarray starts on element *i,* the maximum subarray is the difference between the value at index *i* and the maximum value between *i*+1 and the end of the array.

The absolute maximum subarray possible for element *i* is defined as:

$$MaxSubarray_i = \max_{i+1 \le j \le n} \{A_j\} - \min_{1 \le k \le i} \{A_k\}$$

In simple terms, to find the absolute maximum subarray for element *i*, it is the difference between the maximum for all values after it minus minimum of the set of values at index 1 to index *i*.

The absolute maximum subarray (max_subarray) is maximum for all elements *i* and is defined as:

$$\max\_subarray = \max_{1 \le i \le n-1} \left\{ \max_{i+1 \le j \le n} \{A_j\} - \min_{1 \le k \le i} \{A_k\} \right\}$$

In the algorithm on the previous pages, the first for loop finds the terms:

$$\max_{i+1 \le j \le n} \{A_j\}$$

by iterating through the array starting at the end and working to the front of the array and determining the maximum between elements i+1 through n; the index of the maximum value is also stored in array "MaxPrice_Day".

The second for loop finds the terms:

$$\min_{1 \le k \le i} \{A_k\}$$

by iterating through the array starting at element 1 and going to element n-1 and determines the minimum value between elements 1 and *i*; the index of the minimum value is also stored in array "MinPrice_Day" . This for loop stops at element n-1 because the maximum subarray can not end and start on the same element so it must begin at the latest at the second to last element.

The third for loop calculates the maximum subarrays for each index *i*:

$$\max_{i+1 \le j \le n} \{A_j\} - \min_{1 \le k \le i} \{A_k\}$$

If the value calculated at index *i* is greater than the maximum subarray, it stores this new maximum subarray value as well as the indexes for the starting and end points of this maximum subarray. At the end of this for loop, the algorithm returns the value of the maximum subarray as well as where it begins and ends.

Problem #7

Thief optimization problem

Question 7a: Optimal Substructure

A problem has optimal substructure if the solutions to the problem incorporate optimal solutions to related subproblems. The problem defines that the thief's backpack can hold a maximum weight *M* and that there are *n* possible items that can be stolen. Define a set of objects that has the maximum possible revenue as S, where any object o_i that is part of this optimal is a member of this set ($o_i \in S$). Assume a function "`ThiefMaxRevenue`" that solves this problem takes as input parameters the following:

1. *v* – List of the individual values of the objects
2. *w* – List of the individual weights of the objects
3. *n* – The number of items that can be stolen
4. *M* – Available weight in the backpack.

The initial call for this problem would be:

```
ThiefMaxRevenue(v, w, n, M)
```

This problem has the property that the optimal solution to the entire problem is built off optimal solutions to subproblems. Take the initial case for object *n*; there are two possible cases for this object. It is either in the

$$ThiefMaxRevenue(v,w,n,M) = \begin{cases} ThiefMaxRevenue(v,w,n-1,M): if\ object\ o_n \notin S \\ ThiefMaxRevenue(v,w,n-1,M-w_n) + v_n: if\ object\ o_n \in S \end{cases}$$

In the first case, object *n* (o_n) is not part of the optimal solution. As such, the optimal solution to the original problem is equivalent to the optimal solution for the new subproblem with object o_n removed (i.e. number of objects is now *n*-1). In the second case where object o_n is part of the optimal solution, then the optimal solution to the original problem is equal to the sum of the value v_n of object o_n plus the optimal solution to the new subproblem for *n*-1 objects and maximum available backpack weight of M-w_n (where w_n is the weight of object o_n).

Question 7b-1: Algorithm Description

C# Implementation of Thief Problem

```
static void Q7_Thief_Max_Value(Q7_Thief_Items[] objects, int max_weight, out int[,] stolen_items)
{

    int n = objects.Length-1; //------Subtracting 1 from length due to making array from 1 to n
    int i, j;
    int v, w;
    stolen_items = new int[n + 1, max_weight + 1];

    //------iterate through the weights and set them to zero
    for (j = 0; j <= max_weight; j++)
        stolen_items[0, j] = 0;
```

```
    //----- Initialize the items to zero value and zero weight
    for (i = 0; i <= n; i++)
        stolen_items[i, 0] = 0;

    //------ Build a two dimensional array similar to LCS problem
    for (i = 1; i <= n; i++)
    {
        w = objects[i].weight;
        v = objects[i].value;

        //----- j is the available weight
        for (j = 1; j <= max_weight; j++)
            //---- Check if this item added to an earlier weight is bigger than current value
            if (j - w >=0 && stolen_items[i - 1, j] < stolen_items[i-1, j - w] + v)
                stolen_items[i, j] = stolen_items[i-1, j - w] + v;
            else
                stolen_items[i, j] = stolen_items[i - 1, j];
    }

    Console.WriteLine("The thief can steal a maximum of ${0}.\n", stolen_items[n, max_weight]);

}

static void Q7_Reconstruct_Stolen_Items_List(Q7_Thief_Items[] objects, int[,] stolen_items)
{

    int i = stolen_items.GetLength(0) - 1;
    int j = stolen_items.GetLength(1) - 1;
    int numb_stolen_items =0;
    int[] list_stolen_items = new int[i];

    while (i > 0 && j > 0)
    {
        //------check if item i is part of the optimal solution
        if (stolen_items[i, j] == stolen_items[i - 1, j])
            i--;
        //---- Check if this is the minimum weight for this optimal solution
        else if (stolen_items[i, j] == stolen_items[i, j - 1])
            j--;
        else
        {
            //---- Item i was stolen so decrement weight by weight of object i
            list_stolen_items[numb_stolen_items] = i;
            numb_stolen_items++;
            j -= objects[i].weight;
            i--;
        }
    }

    //---- Print the stolen items.  Print them backwards so they are in ascending order
    Console.Write("The objects that were stolen were: ");
    for (i = numb_stolen_items; i > 0; i--)
    {
        if (i != numb_stolen_items)
            Console.Write(",\t");

        Console.Write(list_stolen_items[i-1].ToString());
    }
    Console.Write(".\n\n");
}
```

The implementation of this algorithm is very similar to that of the Longest Common Subsequence problem. It involves creating a two dimensional matrix of size *n* by *M*. The third for loop ("`for (i = 1; i <= n; i++)`") in the function "`Q7_Thief_Max_Value`" is where the problem is solved. For each iteration of the outer loop, the problem is solved for the subset of items 1 to *i*. The nested if statements inside the inner for loop ("`for (j = 1; j <= max_weight; j++)`") checks whether for a given weight *j*, the solution to the previous set of *i*-1 items can be improved if item *i* is used. Unlike the LCS problem which did not require a trailing pointer, this function must use one because the weight w_i of an item *i* can be greater than 1.

The second function "`Q7_Reconstruct_Stolen_Items_List`" is similar to the reconstruction function from problem #5. It essentially traverses the matrix from the previous function in the reverse order it was created. If for a given cell in the matrix, an equivalent optimal solution is possible by decrementing the weight or the number of items, it does so. Otherwise, it means that the item *i* is part of the optimal solution. It then decrements the number of items by 1 and the current weight *j* by the weight of object *i* (w_i). The last for loop then prints the objects in reverse order since it loaded the array "`list_stolen_items`" in descending order, and it is easier to read the solution if the list is in ascending order.

Question 7b-2: Algorithm Runtime Analysis: This algorithm has two functions. The first generates a matrix that is analyzed in the second function. To find the combined runtime of the entire algorithm, it is possible to analyze each function (e.g. f_1 for the first function and f_2 for the second function) separately and then analyze the combined results.

$$\theta(f_1 + f_2) = \theta(f_1) + \theta(f_2)$$

The first function has three main loops as well as other constant time operations. The first for loop I will refer to as L1; the second loop is L_2, and the third combined loop (including its nested component), I will refer to as L_3. Since the loops are nested, I will refer to its combined runtime as L_1. Hence the runtime of function #1 (f_1) is:

$$\theta(f_1) = \theta(c) + \theta(L_1) + \theta(L_2) + \theta(L_3) + \theta(c)$$

L_1 has an outer loop of n+1 iterations; the second loop (L_2) iterates M+1 times. The third combined loop (L_3) will be the dominant runtime component. Its outer loop runs *n* times and the inner loop runs M times. As such, its runtime is *nM*. Thus, the runtime for function #1 simplifies to:

$$\theta(f_1) = \theta(1) + \theta(n+1) + \theta(M+1) + \theta(n \cdot M) + \theta(c)$$

$$\theta(f_1) = \theta(n \cdot M)$$

The second function (f_2) has similar runtime to the function in problem #5. It has a while loop that runs until one of two iterators (e.g. *i* and *j*) is at 0. Iterator *i* is bounded by *n* while iterator *j* is bounded by *M*. The worst case runtime of this function is when one iterator decrements to 1 and the other to 0. In that case, the runtime would be:

$$\theta(f_2) = \theta(n + M - 1) \in \theta(n + M)$$

The total runtime of the two functions together is:

$$\theta(f_1 + f_2) = \theta(f_1) + \theta(f_2)$$

$$\theta(f_1 + f_2) = \theta(n \cdot M) + \theta(n + M)$$

$$\boxed{\theta(f_1 + f_2) = \theta(n \cdot M)}$$

Appendix #1 - Source Code for Homework #3 Implemented in C#