Problem #1

Specify a greedy strategy to solve the thief backpack problem from homework #3. Prove whether the strategy guarantees an optimal solution to the problem.

Proposed Algorithm: Select the item with the highest value to weight ratio.

Is the Optimal Solution Guaranteed: No

Example:

- Maximum Weight (W): 10
- Number of Objects (n): 3
- Set of Objects O

Object #	Weight (w_i)	Value (v_i)	Value to Weight Ratio
1	6	30	6
2	5	20	4
3	5	20	4

Optimal Solution: Select items #2 and #3. Maximum total value is 40.

Greedy Solution: Select item #1 first since its value to weight ratio is the highest (6). Once this object is selected, no other objects fit in the remaining weight (4) so the total value is 30. This is less than the maximum total value of 40.

Note: This description assumes the thief can only steal the item or not and that he cannot steal a portion of an item (i.e. the items are indivisible).

Problem #2

For the activity selection problem, prove whether choosing the activity with the least number of conflicts always returns an optimal solution.

Answer: No. This greedy strategy is not optimal.

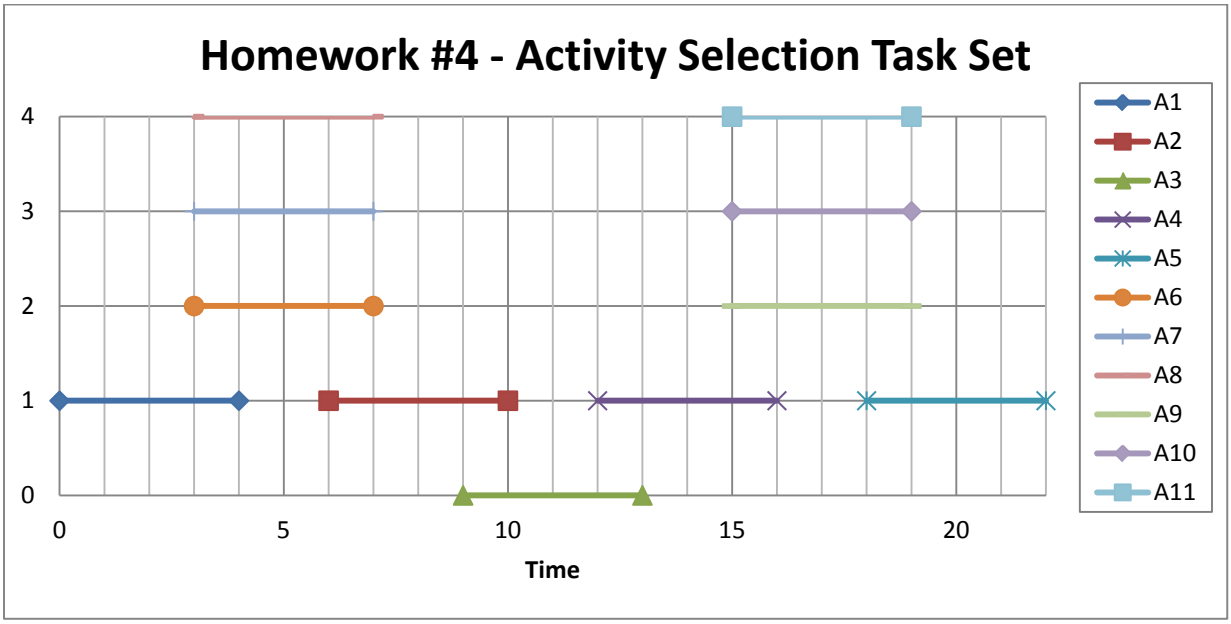
Example: Set of Activities: *A*

Activity#	Start	End	# Conflicts	Conflicting Activities
1	0	4	3	A_6, A_7, A_8
2	6	10	4	A_3, A_6, A_7, A_8
3	9	13	2	A_2, A_4
4	12	16	4	A_3, A_9, A_{10}, A_{11}
5	18	22	3	A_9, A_{10}, A_{11}
6	3	7	4	A_1, A_2, A_7, A_8
7	3	7	4	A_1, A_2, A_6, A_8
8	3	7	4	A_1, A_2, A_6, A_7
9	15	19	4	A_4, A_5, A_{10}, A_{11}
10	15	19	4	A_4, A_5, A_9, A_{11}
11	15	19	4	A_4, A_5, A_9, A_{10}

Table 1 - Least Conflicting Activity Greedy Algorithm Contradictory Example

Optimal Solution (Highlighted in Gray) – Activities A_1, A_2, A_4, A_5

By using the greedy strategy specified in the question, the first activity selected is A_3 (only has 2 conflicts while all others have 3 or more). This selection will then eliminate the optimal solution because activities A_2 and A_4 are not compatible with activity A_3. As such, the maximum number of activities that would be selected using this approach for this set of activities would be three, which is less than the optimal solution of four.



Problem #3 - Exercise 16.1-5 (page 422)

Modified activity-selection problem where each activity A_i has a value v_i. As opposed to the version of activity selection presented in the textbook where the goal was to maximize the total number of activities scheduled, this problem requires that the total value be maximized.

Below is the brute force implementation of this problem done recursively. It is implemented in C# and will serve as the foundation of the dynamic programming problem. An activity is a C# structure (e.g. struct) that has the following data members:

```
struct Activity
{
    public int start;   //---- Start time of Activity
    public int end;     //---- End time of Activity
    public int value;   //---- Value of Activity
}
```

Given *n* activities and a start time and end time for the entire set of activities (i.e. *StartTime* and *StopTime* respective), then for item A_n, there are two possible outcomes. The first case is that A_n is not part of the optimal solution, and the second is that A_n is part of the optimal solution. To determine which is the case, you must determine the maximum value of the two cases as shown in the equation below.

$$ActivitySelection(n, ActivitiesList, StartTime, StopTime) = max \begin{cases} ActivitySelection(n-1, ActivitiesList, StartTime, StopTime) \\ ActivitySelection(n-1, ActivitiesList, StartTime, ActivitiesList[n].start) \\ +ActivitySelection(n-1, ActivitiesList, ActivitiesList[n].end, StopTime) \\ +ActivitiesList[n].value \end{cases}$$

The first case is trivial and involves simply removing activity A_n from the set of activities to be considered. The second case is more complicated. If Activity A_n is part of the optimal solution, then the time consumed by A_n must be removed from the subproblems since it can no longer be reused. As such, two recursive calls are required. The first recursive call determines the optimal solution for the *n*-1 items from the initial start time to the beginning of activity A_n; the second recursive call determines the optimal solution for the n-1 items from the end time of activity A_n to the end of the total available time.

As such, this problem shows overlapping subproblems and optimal substructure making it a candidate for either dynamic programming or a greedy algorithm. No greedy algorithm appears possible for this problem so I went with dynamic programming.

Below is the function prototype for the Brute Force call to this function.

```
static int Q3_Activity_Selection_With_Value_Brute_Force(Activity[] list_of_activities, int n, int seq_start, int seq_end)
```

The input parameters are defined as:

- *list_of_activities* – Set of activities to consider. They are of type "Activity" which is defined above as a C# structure.
- *n* – Number of activities remaining in the recursive subproblem set.
- *seq_start* – Minimum start time for the remaining subproblem set.
- *seq_end* – Maximum end time for the remaining subproblem set.

Below is the initial call and function code for the Brute Force Solution[1].

Initial Call:
```
int activity_results_BF = Q3_Activity_Selection_With_Value_Brute_Force(list_of_activities, Q3_n, 1, max_end_time);
```

```
static int Q3_Activity_Selection_With_Value_Brute_Force(Activity[] list_of_activities, int n, int seq_start, int seq_end)
{

    int activity_not_part_of_sol_val;
    int activity_part_of_sol_val;
    bool n_valid = list_of_activities[n].start >= seq_start && list_of_activities[n].end <= seq_end;

    if (1 == n)
        if (n_valid)
            return list_of_activities[n].value;
        else return 0;

    if (!n_valid) return Q3_Activity_Selection_With_Value_Brute_Force(list_of_activities, n - 1, seq_start, seq_end);

    if (2 == n)
    {
        bool first_valid = list_of_activities[1].start >= seq_start && list_of_activities[1].end <= seq_end;

        //----- Return max of 1 and n if both are valid
        if (first_valid && n_valid)
        {
            //----- Not overlapping case
            if (list_of_activities[1].end <= list_of_activities[n].start || list_of_activities[1].start >= list_of_activities[n].end)
                return list_of_activities[1].value + list_of_activities[n].value;
            //---- Overlapping case
            else return Math.Max(list_of_activities[1].value, list_of_activities[n].value);
        }
        else if (n_valid) return list_of_activities[n].value;
        else if (first_valid) return list_of_activities[1].value;

    }

    //---- Determine the maximum value if item n is NOT part of optimal solution
    activity_not_part_of_sol_val = 0;
    activity_not_part_of_sol_val = Q3_Activity_Selection_With_Value_Brute_Force(list_of_activities, n - 1, seq_start, list_of_activities[n].start);
    activity_not_part_of_sol_val += Q3_Activity_Selection_With_Value_Brute_Force(list_of_activities, n - 1, list_of_activities[n].end, seq_end);
    activity_not_part_of_sol_val += list_of_activities[n].value;

    //---- Determine the maximum value if item n IS part of optimal solution
    activity_part_of_sol_val = Q3_Activity_Selection_With_Value_Brute_Force(list_of_activities, n - 1, seq_start, seq_end);

    return Math.Max(activity_not_part_of_sol_val, activity_part_of_sol_val);

}
```

This problem keeps looking at the maximum activity value based off three primary parameters. They are:

- Items remaining in the set
- Start time of Set
- End Time of Set

Solution: Create a dynamic programming solution that calculates these values in an iterative way to prevent duplication of calculation. This solution requires a three dimension array of integers of size *n* by *MaxEndTime*[2] by *MaxEndTime* (where *MaxEndTime* (*MET*) is the quantized value of maximum end time of all activities).

Below is the solution to this problem using dynamic programming. It requires two functions. The first function "`Q3_Activity_Selection_with_Value_DP`" calculates the three dimensional array which is returned. The second

[1] **Note:** One assumption made in this code for simplification is that the start and end time of the activities is an integer. The problem did not specify that as a condition, but this is possible because the activity start and stop times are quantized. In other word, there is a minimum time unit to express them be it nanoseconds, milliseconds, minutes, days, etc.

[2] Future references to **MaxEndTime** below will be abbreviated as *MET*.

function “Q3_Print_Activity_Selection_DP” rebuilds the list of activities that composed the optimal solution found and prints them to the console.

```
static int[, ,] Q3_Activity_Selection_with_Value_DP(Activity[] list_of_activities, int max_time)[3]
{
    int n = list_of_activities.Length - 1;
    int i;
    int start_time, end_time;
    bool activity_compatible;
    Activity cur_activity;
    int val_with_activity, val_no_activity;
    int[, ,] activity_selection_results = new int[n + 1, max_time + 1, max_time + 1];

    //---- Iterate through all the activities from 1 to n;
    //---- First Outer Loop
    for (i = 1; i <= n; i++)
    {
        cur_activity = list_of_activities[i]; //---- Extract i-th activity

        //----- Iterate through all combinations of start and end times
        //----- Middle Loop
        for (start_time = 1; start_time < max_time; start_time++)
        {
            //---- Inner Loop
            for (end_time = start_time + 1; end_time <= max_time; end_time++)
            {
                //---- Verify current activity is compatible with previous set
                if (cur_activity.start >= start_time && cur_activity.end <= end_time)
                    activity_compatible = true;
                else
                    activity_compatible = false;

                //----- If current activity is not compatible with current start and stop time, take previous value
                if (activity_compatible == false)
                {
                    activity_selection_results[i, start_time, end_time] = activity_selection_results[i - 1, start_time, end_time];
                }
                else
                {
                    //---- Determine the value with and without the task
                    val_with_activity = activity_selection_results[i - 1, start_time, cur_activity.start];
                    val_with_activity += activity_selection_results[i - 1, cur_activity.end, end_time];
                    val_with_activity += cur_activity.value;

                    val_no_activity = activity_selection_results[i - 1, start_time, end_time];

                    activity_selection_results[i, start_time, end_time] = Math.Max(val_with_activity, val_no_activity);
                }
            }
        }
    }

    //----- Return Activity Results Matrix
    return activity_selection_results;
}

static void Q3_Print_Activity_Selection_DP(Activity[] list_of_activities, int[, ,] activity_selection_results, int n, int start_time, int end_time)
{
    Console.Write("An optimal list of activities is = [  ");
    Q3_Print_Activity_Selection_DP_Recursive(list_of_activities, activity_selection_results, n, start_time, end_time);
    Console.WriteLine("]");
}


static void Q3_Print_Activity_Selection_DP_Recursive(Activity[] list_of_activities, int[, ,] activity_selection_results, int n, int start_time, int end_time)
{
    //---- Recursion termination condition
    if (n == 0 || end_time == start_time) return;

    //----- Check if activity i is part of the optimal solution
    if (activity_selection_results[n, start_time, end_time] == activity_selection_results[n - 1, start_time, end_time])
    {
        Q3_Print_Activity_Selection_DP_Recursive(list_of_activities, activity_selection_results, n - 1, start_time, end_time);
        return;
    }

    //----- Check if activity i is part of the optimal solution
    if (activity_selection_results[n, start_time, end_time] == activity_selection_results[n, start_time, end_time - 1])
    {
        Q3_Print_Activity_Selection_DP_Recursive(list_of_activities, activity_selection_results, n, start_time, end_time - 1);
        return;
    }

    //----- item is part of the sequence so print it
    Q3_Print_Activity_Selection_DP_Recursive(list_of_activities, activity_selection_results, n - 1, start_time, list_of_activities[n].start);
    Console.Write("A{0}  ", n);
    Q3_Print_Activity_Selection_DP_Recursive(list_of_activities, activity_selection_results, n - 1, list_of_activities[n].end, end_time);

}
```

[3] Note the code here is written intentionally in small font to allow for all commands not to spill over onto multiple lines. For larger font, please see the appendix.

Algorithm Run Time: The main function "`Q3_Activity_Selection_with_Value_DP`" is composed of three loops that bound its running time. The outer loop runs *n* times; the middle and inner loops run *max_time* times which is equivalent to *MET* mentioned previously. As such the runtime of this algorithm is:

$$\theta(MET^2 \cdot n)$$

The second function "`Q3_Print_Activity_Selection_DP_Recursive`" that prints the list of activities in the optimal solution. Its worst case run time is also when its number of elements is decremented to 1 and its end time is decremented to its initial start time (or vice versa). As such. Its runtime is bounded by $O(MET \cdot n)$.

Combining the total runtime of the first and second functions gives a worst case runtime of

$$\theta(MET^2 \cdot n)$$

If *MET* (MaxEndTime) is a constant or $MET^2 << n$, then this can be simplified to Θ(n).

Algorithm Analysis: The idea for this algorithm became obvious once I implemented the Brute Force solution where the answer to previous problems were based off the subproblems start and end times. For each iteration of the other for loop, I calculate the optimal solution for the first to i^{th} values for all combinations of start and end times. This data then provides all the data required to solve for the first to $(i+1)^{th}$ items. Once the algorithm has completed, the maximum value will be in cell [n, 1, MET (i.e. max_time)] in the array "`activity_selection_results`".

The function "`Q3_Print_Activity_Selection_DP_Recursive`" takes the array "`activity_selection_results`" from function "`Q3_Activity_Selection_with_Value_DP`" and traverses it in the reverse direction it was built. If for a given activity index *i*, start time, and end time, there are three possible cases:

1. Reducing the number of activities by one yields the same maximum value – In this case, the algorithm recursively calls itself with *i* decremented by 1.

2. If not case #1 and by reducing the end time by 1 yields the same maximum value – In this case, the algorithm recursively calls itself with the end time decremented by 1.

3. Item *i* is part of the optimal solution. The algorithm then prints the activities in the optimal solution before activity A_i began and after activity A_i ended.

Problem #4 - Exercise 16.2-4, page 427

Professor Gekko is skating across North Dakota and can travel a maximum distance of *m* miles without stopping to get new supplies. Provide an algorithm to minimize the number of stops.

This problem defines a set *S* of the stops that can be made in the journey. Define the number of stops in this set as *n*. Since the book does not specify the nature of set *S* of *n* stops, so two possible cases must be considered. The first case is that the set is sorted by closest to the starting journey. In that case, the algorithm's runtime does not need to consider the time to sort these values. If the S is unsorted, then it should be sorted from closest to the start to those furthest away. This presorting will reduce the runtime of the algorithm. I will assume the set is sorted, but if it was not, then the run time analysis of the presort algorithm would also need to be considered.

Given the n^{th} item in the set, there are two possible cases. It is either in the optimal solution or it is not. Given a function MinStops that takes the starting pointing, maximum distance without stopping *m*, number of objects *n*, and the set of stops *S*, then the two outcomes is defined as:

$$MinStops(S, n, m, start) = min\begin{cases} MinStops(S, n-1, m, start), item\ n\ not\ in\ optimal\ solution \\ MinStops(S, n-1, m, S[n].location) + 1, item\ n\ is\ in\ optimal\ solution \end{cases}$$

For a Greedy Algorithm to apply, then the globally optimal solution must consist of locally optimal solutions to subproblems. Given a previously picked stop S_i, then the locally optimal solution would be to pick the next stop S_j such that the distance between S_j and S_i is closed to m as possible without exceeding it. This approach then minimizes the size of the resulting subproblem since the distance between S_j and the end of the path is now minimized. Hence, the optimal solution of picking the next stop to be the one closest to distance *m* away from the previous stop without exceeding it leads to the globally optimal solution.

The function below shows the greedy implementation of this problem in a bottom-up style written in C#.

```
static void Q4_MinStops(int[] list_of_possible_stops, int n, int m){

    int previous_stop = 0;
    int i;
    int numb_stops = 0;

    Console.Write("Professor Gekko needs to stop at stops: ");

    //---- Iterate through all the stops
    for(i=0; i<n; i++){
        //-----
        if (list_of_possible_stops[i + 1] - previous_stop > m)
        {
            numb_stops++;
            previous_stop = list_of_possible_stops[i];
            Console.Write("#{0},  ", i);
        }
    }

    Console.Write("\n");
    Console.Write("Professor Gekko's journey required {0} stops.\n\n\n", numb_stops);

}
```

The for loop in the C# implementation sets the bound on the running time. It iterates through each of the *n* stops so the overall runtime of this function is $\Theta(n)$. If the distance of the next stop is greater than *m* distance from the previous location Professor Gekko stopped, then Gekko must stop at index *i* incrementing the number of stops and printing the stop number.

Total runtime of this algorithm would be:

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

If we assume the list of stops is presorted, then the runtime is $\theta(n)$.

Appendix #1 - Source Code for Homework #4 Implemented in C#