Bryan Arnold

CSE 3500

3/27/16

Homework 7

Exercises for DP Algorithms on Sequences

For this problem, the LIS algorithm must be used to determine the longest increasing subsequence in the given array A. Then, to find the answer to the question, we use a dynamic programming table to display the outcomes and whichever has the highest number of increasing order at the end is the answer.

Algorithm:

Input: The array A

For i = 0 to length of A (n)

    Do LIS[i] = 1 //base case

For j = 0 to length of A (n)

For k = 0 to length of A (n)

If A[j] is greater than A[k] as well as LIS[j] is less than LIS[k] + 1

    Set LIS[j] to LIS[j] + 1

End for j

End for k

Display result in some way, table based.

Here is the dynamic programming table for A = {1, 4, 2, 9, 7, 5, 8, 2}:

A	1	4	2	9	7	5	8	2
LIS	1	1	1	1	1	1	1	1
LIS	1	2	2	3	3	3	4	2

In the second LIS row, the number represents the total size of the increasing subsequence possible when ending at that element in the array. Thus, the answer would end at 8, making the LIS size 4. (1, 2, 7, 8) or (1, 2, 5, 8).

Coin Change

1. The greedy algorithm for this problem will always choose the highest denomination of coin value in the given sets of coin change. Although the greedy algorithm would be able to produce some output, it is not the optimal solution. This is because there is a lack of nickels in the type of coins. So, for instance of needs of nickels, pennies would be used instead, making the algorithm not as optimal as it could be. Also, confusions between the 50-cent coin and the quarter can be seen, as in the greedy algorithm two quarters make up 50 cents, not a single coin.
2. Algorithm:

   Input: an array of coin denominations (coins[]; d1 through di) as well as the change we want to get (w). Defined as C(i, w).

   If the amount of change is equal to 0, simply return 0 as the answer since there is no change to begin with. Also, checks if there are no more changes to be made:

   If (w = 0) return 0;

   Next, you would loop over the collection of coin denominations from greatest to least denomination an initialize a coin value variable to the denomination:

   For i from coins[] length to 1
   Set a new variable for coin value to the denomination: value = coins[i – 1]

   If the next largest coin is found, return 1 plus the difference in coin value to the total amount of change desired, as well as displaying the coin value used. In the end, the coins used for the change will be displayed through this:

   If w >= value, display the coin value being used
   Return 1 + C(coins[], w – value

   End if
   End for i

   If there is no solution for the given change, simply display that here and return 0 to end the program.

   }

   Runs in O(nk).

3. Algorithm:

Input: an array of coin denominations (coins[]; d1 through di) as well as the change we want to get (w). Defined as C(i, w).

If the amount of change is equal to 0, simply return 0 as the answer since there is no change to begin with. Also, checks if there are no more changes to be made:

If (w = 0) return 0;

Next, you would loop over the collection of coin denominations from greatest to least denomination an initialize a coin value variable to the denomination:

For i from coins[] length to 1
Set a new variable for coin value to the denomination: value = coins[i – 1]

Now, if the recurrence of the algorithm + 1 is less than the minimum denomination of the coins (coins[] min), add a $1 coin:

```
If w >= value
        If C(coins[], w – value) + 1 is less than the value
                Value = C(coins[], w – value) + 1   //$1 added
                Value = coins[i]
        End if
End if
```

If the next largest coin is found, return 1 plus the difference in coin value to the total amount of change desired, as well as displaying the coin value used. In the end, the coins used for the change will be displayed through this:

```
If w >=         value, display the coin value being used
Return 1 + C(coins[], w – value)

End if
End for i
```

If there is no solution for the given change, simply display that here and return 0 to end the program.

}

Runs in O(nk).

Longest Common Subsequence

To find the longest common subsequence of these two sequences, after doing the algorithm, we set up a dynamic programming table of the two sequences. The row of characters is S1, and S2 is the column of characters. If there is a zero in the table , it means there is no match between the two parts of the sequences. Otherwise, there are two methods in which to fill the spots in the table. First, if the two characters that meet up in a spot are a match, that adds on one to the value in that spot. So, if two A's meet, it would be 1. Following spots would be found out by looking at the diagonal relationship between spots, if the last one was a match, the next diagonal one will add that on to its total. Lastly, the second way a spots value is determined is by if they do not match. If there is no match, add nothing. Here is the dynamic programming table for S1 and S2.

	A	B	A	A	B	B	A
B	0	1	1	1	1	1	1
A	1	1	2	2	2	2	2
A	1	1	2	3	3	3	3
A	1	1	2	3	3	3	4
B	1	2	2	3	4	4	4
A	1	2	3	3	4	4	5
B	1	2	3	3	4	5	5

Now, to find the answer from this table, look at the last cell first. Start there, and now look to the left of it to see if it matches the value of current cell. If it does, move to that cell and repeat the process. If it doesn't match, go upwards right above the cell. Now, repeat the process again of looking left. This is known as backtracking. Keep doing this process until you reach a 0 on the table. The answer will be the pathway created from the 0 to the last cell, based off the circled cell values. For this problem, the path would create the sequence BAABB as the longest common subsequence.

Exercise 2(b)

This algorithm solves the issues in the algorithm in part a) essentially, so the following algorithm is an improvement on how to do this problem. Hi = 0 when I = n to avoid index out of bounds problems.

Algorithm:

Input: The revenue of stressful jobs and the revenue of low-stress jobs per week. (Low-stress revenue: L1 to Ln. High-stress revenue: H1 to Hn).

First, iterate over the number of weeks in the schedule for work:

For i =1 to n

If the high-stress revenue of the next week is greater than high-stress revenue of the current week as well as the high-stress revenue of the next week is greater than a low-stress job this week and the low-stress job revenue of the next week, output either to choose no job in week i and choose a high stress job the next week:

If Hi+1 > Hi and Hi+1 > Li + Li+1

Display: “Choose no job in week I”

Display: “Choose a high-stress job in week i_{i+1}”

i + 2

Otherwise, display choice to choose low-stress job in current week:

Else

Display: “Choose a low-stress job in week I”

i + 1

End if

End for i