COMP 550.002: Fall 2023
Assignment 5

Announced: November 6, 2023

Due Date: November 20, 2023

All problems are collaborative. You can form a group of at most four students to collaborate. You MUST mention the names of your collaborators, and cite any material you took help from (including discussions on canvas by other students) except the textbook and provided slides.

CLRS refers Cormen et al. textbook.

Note: Solving Problems 1–4 prepares you for Midterm 2.

Problem 1 (2 + 10 = 12 Points)

Consider the **coin change** problem discussed in the class. Assume that there are three coin denominations 1 cent, 4 cents, and 6 cents. We want to make a change of 17 cents using the minimum number of coins.

(a) Will the greedy algorithm for solving the coin change problem (introduced in the lectures on greedy algorithms) produce an optimal solution for this coin-change instance? Justify your answer.

(b) In class, we have seen two different recurrence relations (i.e., $change(x)$ and $change(i, x)$) and corresponding bottom-up dynamic programming algorithms. For the coin-change instance given above (i.e., change of 17 cents using 1 cent, 4 cents, nd 6 cents), fill up dynamic programming tables using both dynamic programming approach.

Problem 2 (8 + 12 + 5 = 25 Points)

[**This problem is based on CLRS problems 15.2-2**] The **0-1 knapsack problem** is the following. A thief robbing a store wants to take the most valuable load that can be carried in a knapsack capable of carrying at most W pounds of loot. The thief can choose to take any subset of n items in the store. The i-th item is worth v_i dollars and weighs w_i pounds, where v_i and w_i are integers. Which items should the thief take so that the sum of the values of the taken item is maximized? (This is called the 0-1 knapsack problem because for each item, the thief must either take it or leave it behind. The thief cannot take a fractional amount of an item or take an item more than once. **(Note: This problem description is adapted from CLRS Sec 15.2. Reading the "greedy vs dynamic programming" labelled portion of this section should help answering this and the following problem.)**

(a) Give a recurrence relation for the maximum value of the items the thief can take.

(b) Using the recurrence relation in part (a), give a bottom-up dynamic programming solution for the 0-1 knapsack problem. (You will get full credit if your solution runs in $O(n \cdot W)$ time where n is the number of items and W is the capacity of the knapsack. Note: Like the coin change problem, $O(n \cdot W)$ running time is pseudo-polynomial for this problem.)

(c) Consider the following input for the 0-1 knapsack problem. The capacity of the knapsack is $W = 5$. The number of items is $n = 3$. The weights of these items are $w_1 = 1, w_2 = 2$, and $w_3 = 3$. The values of these items are $v_1 = 120, v_2 = 80$, and $v_3 = 100$. Give the table produced by the dynamic programming solution you gave in (b).

Problem 3 (12 Points)

For the following recurrence relation of $F(i, j)$, write pseudo-code snippet to compute the corresponding dynamic programming table. Here, i ranges from 0 to m and j ranges from 0 to n. You only need to write pseudo-code corresponding to the non-base cases, i.e., the third case in the recurrence relation (not the cases where $F(i, j)$ is 0 or ∞).

$$F(i,j) = \begin{cases} 0 & \text{if } i = 0 \text{ or } i = m\,, \\ -\infty & \text{if } j = n\,, \\ \max(F(i+1,j)+1, F(i,j+1)+1, F(i-1,j+1)+1) & \text{otherwise}\,. \end{cases}$$

(Hint: try to determine the dependency of subproblems and the ordering so that dependent subproblems are solved first.)

Problem 4 (15 Points)

Consider the bottom-up dynamic programming algorithm in CLRS to solve the longest common subsequence (LCS) problem. Give a pseudocode to reconstruct an LCS from the completed c table and the original sequences $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_m \rangle$ in $O(m+n)$ time, without using the b table. (Hint: trace back the c table, For a cell $c[i, j]$, try to determine which cell is responsible for the value of $c[i, j]$.)

Problem 5 (8 + 8 = 16 Points)

Consider the graph shown in Fig. 1.

(a) Show the d and π values that result from running breadth-first search on G in Fig. 1, using vertex w as the source. Assume that neighbors of a vertex are visited in alphabetical order.

(b) Show the d, f, and π values that result from running depth-first search on G in Fig. 1. Assume that the `DFS-Visit` procedure is first called for vertex w (i.e., w is the first source). Assume that neighbors of a vertex are visited in alphabetical order. Classify all edges into tree, back, forward, and cross edges. Show the classification by labelling each edge accordingly.

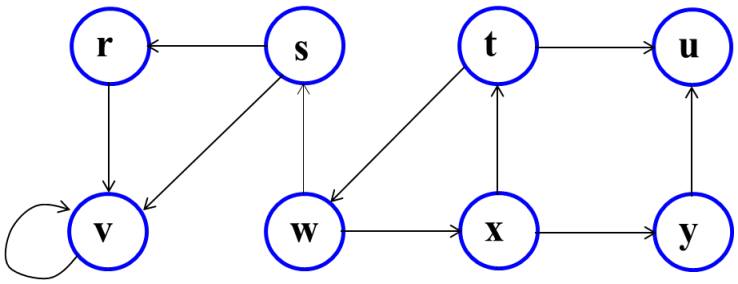

Figure 1: Graph G

Problem 6 (20 Points)

Implement the bottom-up dynamic programming algorithm to solve the longest common subsequence problem. Details to be provided in a separate document.

Problem 8 (Extra Credit: 20 Points)

The edit distance between two strings is the minimum number of letter insertions, letter deletions, and letter substitutions required to transform one string into the other. For example, given two strings SUNNY and SNOWY, we need at least three operations to transform SUNNY into SNOWY (and vice versa). To illustrate this, let's consider the transformation of SUNNY into SNOWY. One way to transform SUNNY into SNOWY is (i) delete 'U', (ii) substitute the second 'N' in SUNNY by 'O', and (iii) insert 'W' between newly substituted 'O' and 'Y'. The transformation from SNOWY into SUNNY can be done by the same sequence of operations with each operation reverted (i.e., (i) insert 'U', (ii) substitute 'O' by 'N', and (iii) delete 'W').

```
S – N O W Y        – S N O W – Y
S U N N – Y        S U N – – N Y
Cost: 3            Cost: 5
```

Figure 2: Alignment of SUNNY and SNOWY

Edit distances can be better visualized by considering alignment of the strings. Technically, an alignment is simply a way of writing the strings one above the other. Fig. 2 shows two possible alignments of SUNNY and SNOWY. The '–' indicates a "gap"; this represents insertion/deletions (e.g., in the left alignment in Fig. 2, the alignment of '–' and 'U' means that we need to insert 'U' to make SUNNY from SNOWY and delete 'U' to make SNOWY from SUNNY). The cost of an alignment is the number of columns in which

the letters differ. Thus, the first alignment's cost is 3, while the latter one's cost is 5. For these two strings, there is no possible alignment that costs less than 3, implying that the edit distance is 3.

Now, given two strings $X[1:m]$ and $Y[1:n]$, write a dynamic programming algorithm to compute the edit distance between X and Y. Also, write the corresponding recurrence relations behind your dynamic programming algorithm. (As mentioned in class and textbook, you should first try to determine an optimal substructure and recursive structure of the problem. As a hint: similar to the LCS problem, can we determine the edit distance of $X[1:i]$ and $Y[1:j]$ (prefixes of X and Y) using edit distances of $X[1:i-1]$ and $Y[1:j]$, $X[1:i]$ and $Y[1:j-1]$, and $X[1:i-1]$ and $Y[1:j-1]$?)