CSCI 270 Problem Set 4 Solutions

Fall 2025

Problem 1. *You are given an $m \times n$ grid of integers, where each cell (i, j) contains a score $a[i, j]$. Starting from the bottom-left corner $(0, 0)$, you want to move to the top-right corner (m, n). At each step, you are only allowed to move* ***up*** *or* ***right****. Please design a polynomial time algorithm to find the maximum total score that can be collected along such a path. Please provide the pseudocode, running time, and correctness proof.*

Solution. We use the following pseudocode.

```
1  def soln(A):
2      m, n = len(A) - 1, len(A[0]) - 1
3      dp = [[0] * (n + 1) for _ in range(m + 1)]
4
5      for i in range(m, -1, -1):
6          for j in range(n, -1, -1):
7              if i == m and j == n:
8                  dp[i][j] = A[i][j]
9              elif i == m:
10                 dp[i][j] = A[i][j] + dp[i][j + 1]
11             elif j == n:
12                 dp[i][j] = A[i][j] + dp[i + 1][j]
13             else:
14                 dp[i][j] = A[i][j] + max(dp[i + 1][j], dp[i][j + 1])
15
16     return dp[0][0]
```

The algorithm runs in time $O(nm)$, as line 2 is constant time, line 3 requires $O(nm)$ to populate `dp`, and lines 5 – 14 loop over $O(nm)$ many (i, j) pairs, with constant work performed for each such pair.

To see that the algorithm is correct, we claim that the `for` loops populate each entry `dp[i][j]` with the maximum total score that can be achieved when traveling from (i, j) to (m, n), using only upward and rightward steps. This is true for the pair (m, n), as the score collected at (m, n) itself equals $A[m][n]$. (See lines 7 – 8.) It holds for all pairs of the form (m, j) with $j < m$, owing to the fact that only upward steps can be made from the rightmost column. This corresponds to the assignment on line 10; notably, `dp[i][j + 1]` will be correctly populated owing to the fact that `j` is processed in decreasing order. By similar reasoning, `dp` is correctly populated for entries of the form (i, n) with $i < m$, as only rightward steps can be mode from the topmost column of the grid. (Again, note that `dp[i + 1][j]` takes the correct value on line 12, as `i` is processed in decreasing order.)

Finally, it holds for all remaining pairs (i, j), as the maximum score from (i, j) is achieved by collecting $A[i][j]$ and taking either a rightward step to $(i+1, j)$ or an upward step to $(i, j+1)$, based upon which one yields a greater total payoff to (m, n). This is implemented by line 14; note once more that both entries of `dp` on the right hand side will be correctly populated owing to the fact that `i` and `j` are processed in decreasing order. $\square$

Problem 2. *Given a string $S = (a_1, \ldots, a_m)$ and a list of strings $A_1, \ldots, A_n$. We assume that each string A_j is a uniquely-identified substring of S. That is, for each A_j, there is a unique pair (i_1, i_2) such that $A_j = (a_{i_1}, \ldots, a_{i_2})$.*

We say that a tuple of strings $(A_{j_1}, \ldots, A_{j_r})$ is a reconstruction of S if their concatenation is exactly S. For example, if $S = abcdefg$ and $A_1 = cde$, $A_2 = ab$, and $A_3 = fg$, then (A_2, A_1, A_3) is a reconstruction of S.

Now, given a string $S = (a_1, \ldots, a_m)$ and a list of strings $A_1, \ldots, A_n$, can you design an algorithm to compute how many reconstructions of S can be obtained from $A_1, \ldots, A_n$? For example, if $S = abcdefg$ and $A_1 = defg$, $A_2 = abc$, $A_3 = de$, $A_4 = fg$, $A_5 = abcde$, then you can obtain two reconstructions of S: namely, (A_2, A_1) and (A_5, A_4). Please provide the pseudocode, running time, and correctness proof.

Solution 1 (Can repeat substrings). We use the following pseudocode.

```
def soln(S, A):
    for (j, Aj) in enumerate(A):
        L, R = unique starting and ending indices of Aj in S
        A[j] = (Aj, L, R)

    m = len(S)
    dp = [0] * (m + 1)
    dp[0] = 1

    ends_at = [[] for _ in range(m + 1)]
    for (Aj, L, R) in substrings:
        ends_at[R].append(L)

    for i in range(1, m + 1):
        for L in ends_at[i]:
            dp[i] += dp[L - 1]

    return dp[m]
```

The algorithm runs in time $O(mn)$, dominated by the time required to deduce the starting and ending indices of each substring A_j within S.[1] Otherwise, the algorithm performs $O(m)$ work on lines $6-8$, $O(m+n)$ work on lines $10-12$, and again $O(m+n)$ work on lines $14-16$, as the total runtime required by the inner `for` loop across all iterations of the outer `for` loop is $\sum_i |\text{ends\_at[i]}| = n$.

We now argue that the algorithm is correct. We claim that at its conclusion, `dp[i]` equals the number of reconstructions of the string $S_{\leq i} = (a_1, \ldots, a_i)$. This certainly holds for `dp[0]` on line 8, as there is only one reconstructions of the empty string. (Namely, the 0-tuple containing no substrings.) Now suppose the first $i-1$ entries of `dp` are correct and consider lines $15-16$. In these, the program iterates through all substrings A_j of S which end at index i, and increments `dp[i]` by `dp[L - 1]` where L is the starting index of A_j. By our inductive hypothesis, `dp[L - 1]` equals the number of reconstructions of $S_{L-1} = (a_1, \ldots, a_{L-1})$. Thus `dp[L-1]` equals exactly the number of reconstructions of S_i which conclude with the substring A_j. As line 15 iterates through all such tasks A_j ending at i, this tracks precisely the total number of reconstructions of S_i. □

Solution 2 (Cannot repeat substrings). **The previous solution was incorrect. Use only Solution 1 for now.** □

Problem 3. *You are given n items, each with a weight w_i and a value v_i. You are also given a knapsack with a maximum capacity W. The goal is to determine the maximum total value that can be obtained by selecting a subset of these items, subject to the constraint that the total weight does not exceed W. Each item can either be taken or not taken; that is, you cannot take a fraction of an item to get a partial value.*

For example, if there are 4 items with weights and values $(2,3),(3,2),(4,4),(5,6)$ and the knapsack capacity is $W = 9$, then you can take the last two items $(4,4)$ and $(5,6)$ to get a total value of 10.

Please provide a dynamic programming algorithm with pseudocode, running time analysis, and a correctness proof.

Solution. We use the following pseudocode.

```
def soln(weights, values, W):
    n = len(weights)
    dp = [[0] * (W + 1) for _ in range(n + 1)]

    for i in range(1, n + 1):
        for w in range(0, W + 1):
            if weights[i - 1] <= w:
                dp[i][w] = max(
                    dp[i - 1][w],
```

[1]Strictly speaking, this can be done in time $O(m + \sum_j |A_j|)$, using the Aho–Corasick algorithm.

```
10                     dp[i - 1][w - weights[i - 1]] + values[i - 1]
11                 )
12             else:
13                 dp[i][w] = dp[i - 1][w]
14
15     return dp[n][W]
```

The algorithm runs in time $O(nW)$, as constant work is performed within the innermost `for` loop and initialization of `dp` requires time $O(n)$.

For correctness, we claim that at the conclusion of the algorithm, for each $i \in [n]$ and $w \in [W]$, `dp[i][w]` equals the maximum value that can be attained by selecting from among the first i items subject to total weight at most w. We induct on pairs (i, w) with respect to the dictionary order. Correctness clearly holds for `dp[0][0]` at initialization on line 3. Fix an (i, w), and suppose that `dp` has been correctly populated for all entries (i', w') which are strictly less than (i, w) under the dictionary order. Then `dp[i][w]` will be correctly populated on lines 7 – 13, as any optimal strategy for selecting from the first i items with weight limit w either: 1) selects the ith item with weight w_i and selects optimally from among the first $i-1$ items with weight limit $w - w_i$, assuming $w_i \leq w$ or 2) selects optimally from the first $i-1$ items with weight limit w, if $w_i > w$. As all terms on the right side of lines 8 – 13 are strictly less than (i, w) under the dictionary order, they are correctly populated due to our inductive hypothesis. This completes the argument. $\square$