CSCI 570 Fall 2025
Homework 7

Problem 1

Jack has gotten himself involved in a very dangerous game called the octopus game where he needs to pass a bridge which has some unreliable sections. The bridge consists of 3n tiles as shown below. Some tiles are strong and can withstand Jack's weight, but some tiles are weak and will break if Jack lands on them, leading to a fatal fall. We have been given this information in an array called BadTile(3,n) where **BadTile (j, i) = 1 if the tile is weak and 0 if the tile is strong**. At any step Jack can move either to the tile exactly in front of him (i.e. from tile (j, i) to $(j, i + 1)$), or diagonally to the left or right (if they exist). (No sideways or backward moves are allowed and one cannot go from tile $(1, i)$ to $(3, i + 1)$ or from $(3, i)$ to $(1, i + 1)$). Jack is allowed to start in any (strong) tile $(j, 1)$ in the beginning and end at any (strong) tile (j, n) at the end.

Use dynamic programming to find out how many ways (if any) there are for Jack to pass this deadly bridge. In Fig. 1, we show an example of bad tiles in gray and one of the possible ways for Jack to safely cross the bridge alive.

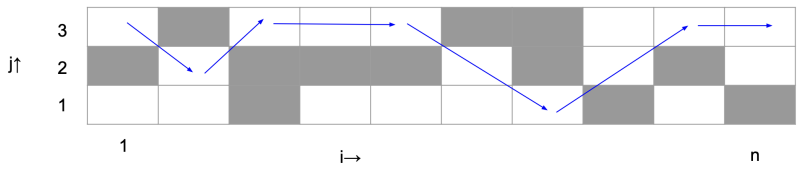

Figure 1: Example of octopus game.

1. Define (in plain English) subproblems to be solved. (2 points).
2. Write a recurrence relation for the subproblems. (3 points)
3. Using the recurrence formula in part b, write pseudocode using iteration to compute the total number of ways to safely cross the bridge. (4 points). Make sure you specify:
   - Base cases and their values.
   - Where the final answer can be found (e.g. opt(n), or opt(0,n), etc.).

Solution

1. OPT(j, i) = The number of ways Jack can reach the block (j, i) safely from the bridge start
2. $$\text{OPT}(j,i) = \begin{cases} 0 & \text{if BadTile(j, i)} = 1 \\ \text{OPT}(j,i-1) + \text{OPT}(j+1,i-1) & \text{if } j = 1 \\ \text{OPT}(j,i-1) + \text{OPT}(j+1,i-1) + \text{OPT}(j-1,i-1) & \text{if } j = 2 \\ \text{OPT}(j,i-1) + \text{OPT}(j-1,i-1) & \text{if } j = 3 \end{cases}$$

Algorithm 1 Computing the total number of ways to safely cross the bridge

3.
```
Given BadTile array of size (3,n)
Initialize OPT array of size (3,n) with 0 for each element
if BadTile(1,1) = 0 then
    OPT(1,1) ← 1
else
    OPT(1,1) ← 0
end if
if BadTile(2,1) = 0 then
    OPT(2,1) ← 1
else
    OPT(2,1) ← 0
end if
if BadTile(3,1) = 0 then
    OPT(3,1) ← 1
else
    OPT(3,1) ← 0
end if
for i = 2 to n do
    for j = 1 to 3 do
        if BadTile(j, i) = 1 then
            OPT(j, i) ← 0
        else if j = 1 then
            OPT(j, i) ← OPT(j, i − 1)+ OPT(j + 1, i − 1)
        else if j = 2 then
            OPT(j, i) ← OPT(j, i − 1)+ OPT(j + 1, i − 1)+ OPT(j − 1, i − 1)
        else if j = 3 then
            OPT(j, i) ← OPT(j, i − 1)+ OPT(j − 1, i − 1)
        end if
    end for
end for
return  OPT(1, n)+ OPT(2, n)+ OPT(3, n)
```

4. $O(N)$.

Problem 2

Given n balloons, indexed from 0 to $n-1$. Each balloon is painted with a number on it represented by array $nums$. You are asked to burst all the balloons. If the you burst balloon i you will get $nums[left(i)] \times nums[i] \times nums[right(i)]$ coins. Here left(i) and right(i) are adjacent indices of i AT THE TIME OF BURSTING (as explained next). After a balloon is burst, the balloons to its left and right become adjacent. We have $nums[-1] = nums[n] = 1$ wherever needed for computing the no. of coins —they are not real balloons, therefore you cannot burst them. Design a dynamic programming algorithm to find the maximum no. of coins you can collect by bursting the balloons wisely. Analyze the running time of your algorithm.

Example: If you have the $nums = [3, 1, 5, 8]$. The optimal solution would be 167, where you burst balloons in the order of 1, 5, 3 and 8. The array $nums$ after each step is:

$$[3, 1, 5, 8] \rightarrow [3, 5, 8] \rightarrow [3, 8] \rightarrow [8] \rightarrow []$$

The total number of coins is $3 \times 1 \times 5 + 3 \times 5 \times 8 + 1 \times 3 \times 8 + 1 \times 8 \times 1 = 167$.

1. Define (in plain English) subproblems to be solved. (2 points)
2. Write a recurrence relation for the subproblems (3 points)
3. Using the recurrence formula in part b, write pseudocode to find the solution.(2 points)
4. Make sure you specify (2 points)
   (a) Base cases and their values.
   (b) Where the final answer can be found.
5. What is the complexity of your solution? (1 point)

Solution

Algorithm 2 Balloon Burst Max Coins

```
1: Create a table dp((n+2)*(n+2)) with all 0s in the table.
2: nums = [1]+nums+[1]
3: for length from 1 to n do
4:    for left from 1 to n-length+1 do
5:       right = left + length-1
6:       for k from left to right do
7:          dp[left][right] = max{dp[left][right], nums[left-1]·nums[k]·nums[right+1]+dp[left][k-1]+dp[k+
            1][right]}
8:       end for
9:    end for
10: end for
11: return dp[1][n]
```

Let dp[i][j] be the maximum coins gained from bursting all the balloons between index i(left) and j(right) in the original array, i and j are included. dp is a two-dimensional array.
The base case is when left==right, then return 0. There are no balloons in the middle to burst. To get the maximum value we can get for bursting all the balloons between [i,j], we just loop through each balloon between these two indices [i,j] and make them to be the last balloon to be burst and choose the one with maximum coins gained. Let the index of last balloon to be burst is k between [i,j],

$$dp[i][j] = \max\{dp[i][j], nums[i-1] \cdot nums[k] \cdot nums[j+1] + dp[i][k-1] + dp[k+1][j]\}$$

$nums[i-1] \cdot nums[k] \cdot nums[j+1]$ is the coined gained when k is the last balloon to be burst. dp[i][k-1] is the maximum coins gained on the left list, and dp[k+1][j] is the maximum coins gained on the right list. FOR each index k between i and $j (i \leq k \leq j)$, we need to find and choose a value of k(last balloon to be burst) with maximum coins gained, and update dp array. In the end, we want to find out dp[0][n-1], which is the maximum value we can get when we burst all the balloons between [0, n-1].

The pseudo-code is shown as above. In the pseudo-code, we added $nums$[-1] and $nums$[n] to $nums$. So the return will be dp[1][n] due to the index change. The time complexity is $O(n^3)$.

Problem 3

We are given an array A of size n and an array B of size m where $n \geq m$. Our goal is to remove $n - m$ elements from A, in such a way that removing them will transform A to B. Notice that the relative order stays the same for the remaining elements.
As an example, let $A = [3, 1, 1, 4, 4, 1]$ and $B = [3, 1, 4, 1]$, then there are 4 ways to remove the elements:

- $[3, 1, 1, 4, 4, 1]$,
- $[3, 1, 1, 4, 4, 1]$,
- $[3, 1, 1, 4, 4, 1]$,
- $[3, 1, 1, 4, 4, 1]$.

Design a dynamic programming algorithm to find out the number of different ways to remove elements from A that will transform A to B (no. of ways $= 0$) if this is not possible. To qualify full credit, your algorithm needs to run in $O(mn)$ time.

1. Define (in plain English) sub-problems to be solved. (3 pts)
2. Write a recurrence relation for the sub-problems and specify the base cases. (5 pts)
3. Analyze its runtime complexity and explain why. (2 pts)

Solution

1. Let $dp[i, j]$ be the number of ways to remove elements from $A[0..i]$ so that it becomes $B[0..j]$. Here we let $X[0..i]$ denote the subarray formed by the **first** i **elements** of X, therefore $X[0..0]$ is the empty array.
2. (a) If $A[i-1] = B[j-1]$ (the last element of $A[0..i]$ is the same as the last element of $B[0..j]$) then $dp[i, j] = dp[i-1, j-1]$(keep $A[i-1]$) + $dp[i-1, j]$(remove $A[i-1]$), otherwise $dp[i, j] = dp[i-1, j]$(remove $A[i-1]$).
   (b) Base cases: $dp[i, j] = 0$ for all $i < j$, $dp[i, 0] = 1$ for all $0 \leq i \leq n$.
3. Initialize the dp array using the above base case, and calculate from bottom up. There are $O(nm)$ elements in the dp table and each depends on constant number of elements, therefore the time complexity is $O(nm)$. And we can find the final answer in $dp[n, m]$.

Problem 4

You are given an integer array A of size N, where each element in the array satisfies $1 \leq A[i] \leq N$. The array may contain duplicate numbers. Your task is to determine the length of the longest **consecutive subsequence** that can be formed from A.

A **subsequence** is a sequence derived from A by removing some elements (possibly none) while keeping the order of the remaining elements unchanged. A **consecutive subsequence** is a special type of subsequence where the numbers appear in strictly increasing and consecutive order. In other words, it should be of the form $[x, x+1, x+2, \dots]$ for some integer x, meaning that every number in the sequence must be exactly one more than the previous number.

FOR example, consider the array $A = [1, 2, 6, 4, 3, 2]$. The sequence $[2, 6, 3]$ is a valid subsequence but **not** a consecutive subsequence because 6 does not equal to $2 + 1$. On the other hand, $[1, 2]$ is a valid consecutive subsequence because all the numbers appear in strictly increasing consecutive order.

Your goal is to design an efficient **dynamic programming algorithm** to compute the **maximum length** of any consecutive subsequence in A. Given that N can be as large as 10^5, your solution should be optimized to run efficiently within these constraints.

1. Define (in plain English) sub-problems to be solved. (2 points)
2. Write a recurrence relation for the sub-problems. (3 points)
3. Using the recurrence formula in part b, write an iterative pseudo-code to find the solution. Make sure you specify base cases. (2 points)
4. What is the complexity of your solution? (1 points)

Solution

1. Let OPT$[i, n]$ be the longest consecutive subsequence of the first i elements of the input array that ends at n.
2. If $n = A[i]$, OPT$[i, n] = OPT[i-1, n-1] + 1$, otherwise OPT$[i, n] = OPT[i-1, n]$. Base case OPT$[0, n] = 0$ for all $1 \le n \le N$.
3. Notice that, even though we have an N^2 dp table, every row differs by only one element, so we can implement the algorithm in the following way:

Algorithm 3 Longest Consecutive Subsequence

```
Initialize dp[N + 1] ← 0
for i ← 1 to N do
    dp[A[i]] ← dp[A[i] − 1] + 1
end for
maxLength ← 0
for i ← 1 to N do
    maxLength ← max(maxLength, dp[i])
end for
return maxLength
```

4. Complexity is $\Theta(n)$.
5. Alternatively, you can fill the entire dp table and yield an $O(N^2)$ algorithm, which is also acceptable.

Ungraded Problems

You are given $n + 1$ rooms, labeled from Room 0 to Room n. There is a heater in Room 0, which warms all rooms from Room 0 to Room r, where r is an unknown integer satisfying $1 \le r \le n$. This means that any room with an index less than or equal to r is warm, while any room with an index greater than r is cold. To determine the exact value of r, you have k thermometers. Each time you bring a thermometer into a

room, it provides feedback based on the room's temperature. If the room is warm, the thermometer remains functional and can be used again in another test. However, if the room is cold, the thermometer breaks immediately and can no longer be used. Your objective is to find the exact value of r while minimizing the total number of times you enter a room. Since thermometers are limited and can break, an efficient strategy is necessary to optimize the testing process. The challenge is to develop a dynamic programming algorithm that determines the minimal number of room entries required to accurately find r.

1. Write down the recursive formula and the meaning of each part.
2. What is the time complexity of the algorithm?

Solution

1. $dp(k, n)$ is the optimal count when there are k thermometers and n floors.

$$dp(k, n) = 1 + \min_{1 \le x \le n} (\max(dp(k-1, x-1), dp(k, n-x)))$$

   $dp(k-1, x-1)$ is the state when the thermometer is broken.
   $dp(k, n-x)$ is the state when the thermometer works well. x denotes the Room x.

2. $O(kn^2)$, or $O(kn \log n)$ [if use binary search]

Assume a truck with capacity W is loading. There are n packages with different weights, i.e. $(w_1, w_2, \ldots w_n)$, and all the weights are integers. The company's rule requires that the truck needs to take packages with exactly weight W to maximize profit, but the workers like to save their energies for after work activities and want to load as few packages as possible. Assuming that there are combinations of packages that add up to weight W, design an algorithm to find out the minimum number of packages the workers need to load.

1. Define (in plain English) subproblems to be solved.
2. Write a recurrence relation for the subproblems.
3. Using the recurrence formula in part b, write pseudocode using iteration to compute the minimum number of packages to meet the objective.
4. Make sure you specify base cases and their values; where the final answer can be found. (2 points)
5. What is the worst-case runtime complexity? Explain your answer.

Solution

1. OPT(w, k) = min packages needed to make up a capacity of exactly w, by considering only the first k packages
2. If $w \ge w_k$, OPT$(w, k) = \min\{1 + \text{OPT}(w - w_k, k-1), \text{OPT}(w, k-1)\}$.
   Otherwise, OPT$(w, k) = \text{OPT}(w, k-1)$.
3. - Initialize for base cases (mentioned below for clarity)
   - For w = 0 to W
     For k = 0 to n //can switch the inner-outer loops
     If not base case: call recurrence
   - Return ans (mentioned below for clarity)
4. OPT$(w, 0) = \infty$ for all $w < 0$ (up to $-W$)
   OPT$(0, 0) = 0$
   OPT$(w, k) = \infty$ for $w < 0$ and all k
   (Not required if the recurrence has the second case to ensure w never takes negative values). The final answer can be found at OPT(W, n)
5. $O(Wn)$