CS 3510 Section B: Design & Analysis of Algorithms October 11th 2024

HW 5: Dynamic Programming

YOUR NAME HERE Due: October 11th 2024

- Please type your solutions using LaTeX or any other software. Handwritten solutions will not be accepted.
- If we ask for a specific running time, a correct solution achieving it will receive full credit even if a faster solution exists. If no target running time is provided, faster solutions will earn more credit.

Problem 1: Road Trip *(25 points)*

You'd like to take a road trip from position 0 to position c_n, but your car can only go 300 miles at a time before needing to refuel. Fortunately, there are n checkpoints numbered 1 to n at specific positions on your path where you can refuel, $C = [c_1, c_2, \ldots, c_n]$, which you can think of as points on a number line. As such, C will have **distinct positive values and will be sorted** and the last checkpoint will always be at c_n. However, each checkpoint at position c_i has an associated "penalty" p_i that it will cost you to use its gas pump (curiously, this is a one-time fee per checkpoint, and does not depend on how much gas you get) and these values will be given as an array $P = [p_1, p_2, \ldots, p_n]$. c_n is always reachable by some series of checkpoints, and we always refuel in the checkpoint located at position c_n. We want to know the minimum total penalty to get to (and refuel at) to position c_n from position 0.

For example, if we have $C = [100, 250, 400]$, and $P = [12, 20, 3]$, then we can go from position 0 to checkpoint 1 at position c_1 and then to checkpoint 3 at position c_3 for a total penalty of 12 + 3 = 15, which is the minimum penalty to get to c_3. We could travel further initially by going from position 0 straight to c_2 (and then to c_3), but this isn't optimal as it costs us total penalty 20 + 3 = 23.

Design a dynamic programming algorithm to efficiently find the minimum total penalty it will cost you to get to to c_n from position 0 given C and P.

(a) Define the entries of your table in words. E.g. $T[i]$ or $T[i][j]$ is ...

> **Solution:** T[i] = min amount of penalty needed to travel from position 0 to C[i].

(b) State base cases and recurrence for entries of the table in terms of smaller subproblems. Briefly explain in words why it is correct.

> **Solution:** The base cases are that T[0] = 0, T[1] = c1. The recurrence: T[i] = min(T[i], T[i-1] + pi, T[j] + pi), for all j = 1 to i-1, if ci - cj ¡= 300.

(c) Write **pseudocode** for your algorithm to solve this problem.

```
n = length(C)
create array DP of size (n + 1)
DP[0] = 0
for i = 1 to n:
  DP[i] = infinity

DP[1] = DP[0] + P[0] // Initialize for the first checkpoint

for i = 2 to n:
  DP[i] = min(DP[i], DP[i - 1] + P[i]) // Consider reaching from previous checkpoint
  for j = 1 to i - 1:
    if C[i] - C[j] <= 300:
```

```
            DP[i] = min(DP[i], DP[j] + P[i]) // Consider reaching from earlier checkpoints

    return DP[n]
```

(d) Analyze the running time of your algorithm.

Solution: Since we are calculating every single min penality all the way up to n, and each run takes from i = 2 to n. it will be $O(n) * O(n)$ which is $O(n^2)$

Problem 2: Difference Score *(25 points)*

You are given two strings X and Y of length n and m respectively. You want to find a difference score D_s which is defined to be the minimum number of steps to transform X to Y using the following options for each step:

1. Replace a character in the string with another character.
2. Add a character to the string.
3. Remove a character from the string.
4. Swap two adjacent characters.

Design an algorithm to compute D_s for two input strings.

Example: Given X = WCA and Y = WABC, $D_s = 2$. (WCA $\rightarrow$ (swap) WAC $\rightarrow$ (insert B) WABC)

(a) Define the entries of your table in words. E.g. $T[i]$ or $T[i, j]$ is ...

> **Solution:** Create a 2D table dp where dp[i][j] represents the minimum number of operations required to transform the first i characters of string X into the first j characters of string Y.

(b) State base cases and recurrence for entries of the table in terms of smaller subproblems. Briefly explain in words why it is correct.

> **Solution:** dp[0][0] = 0 , dp[i][0] = i , dp[0][j] = j

(c) Write **pseudocode** for your algorithm to solve this problem.

```
n = length(X)
m = length(Y)
dp = array of size (n+1) * (m+1)

for i from 0 to n:
    dp[i][0] = i
for j from 0 to m:
    dp[0][j] = j

for i from 1 to n:
    for j from 1 to m:
        if X[i-1] == Y[j-1]:
            dp[i][j] = dp[i-1][j-1]
        else:
            dp[i][j] = dp[i-1][j-1] + 1
```

```
            dp[i][j] = min(dp[i][j], dp[i][j-1] + 1)  // Insert
            dp[i][j] = min(dp[i][j], dp[i-1][j] + 1)  // Remove

            if i > 1 and j > 1 and X[i-1] == Y[j-2] and X[i-2] == Y[j-1]:
                dp[i][j] = min(dp[i][j], dp[i-2][j-2] + 1)  // Swap

    return dp[n][m]
```

(d) Analyze the running time of your algorithm.

Solution: initalizing the array takes O(m*n) runtime, the base case each takes O(m) and O(n) respectively, the double for loop takes O(m*n) runtime because everything inside is able to be done in O(1) run time. The total run time will be O(m*n)

Problem 3: Maximum Pyramid Bananas *(25 points)*

A monkey is placed in given a pyramid of bananas stacks A, where $A[i][j]$ represents the number of bananas in the stack at position (i, j). The monkey starts at position $(0, 0)$, and at each step, he can only go directly down-right or down-left to get to another stack. Additionally, for each down-right move that the monkey makes, he also obtains $bonus_r$ amount of bananas, and for each down-left move that the monkey makes, he also obtains $bonus_l$ amount of bananas. Design an algorithm to find the maximum amount of bananas the monkey can earn from the top of the pyramid to the bottom row. Consider the following example with $bonus_l = 1$ and $bonus_r = 2$:

$$\begin{array}{c} 3 \\ 7 \quad 4 \\ 2 \quad 4 \quad 6 \\ 8 \quad 5 \quad 9 \quad 3 \end{array}$$

(In array form, the input is given as A = [[3], [7, 4], [2, 4, 6], [8, 5, 9, 3]])

The monkey's best path through this pyramid would be $(0, 0), (1, 0), (2, 1), (3, 2)$. It would collect $3 + 7 + 4 + 9 = 23$ bananas from the stacks and $1 + 2 + 2 = 5$ from the moves. The total amount of bananas obtained about therefore be be $23 + 5 = \$28$.

(a) Define the entries of your table in words. E.g. $T[i]$ or $T[i, j]$ is ...

> **Solution:** We'll use a 2D array dp where dp[i][j] stores the maximum bananas the monkey can collect when reaching the stack at position (i, j).

(b) State base cases and recurrence for entries of the table in terms of smaller subproblems. Briefly explain in words why it is correct.

> **Solution:** dp[0][0] = A[0][0] For any position (i, j), the monkey can reach it either from (i-1, j-1) or (i-1, j) $dp[i][j] = max(dp[i-1][j-1] + bonus_l + A[i][j], dp[i-1][j] + bonus_r + A[i][j])$

(c) Write **pseudocode** for your algorithm to solve this problem.

```
n = len(A)
dp = [len(row) for row in A]  set to 0.
dp[0][0] = A[0][0]

for i in range(1, n):
 for j in range(len(A[i])):
  if j > 0
    down_left = 0
  else
    down_left = 0 = dp[i-1][j-1] + bonus_l + A[i][j]
```

```
    if j < len(A[i-1])
      down_right = 0
    else
      down_right = dp[i-1][j] + bonus_r + A[i][j]
    dp[i][j] = max(down_left, down_right)

  return max(dp[n-1])
```

(d) Analyze the running time of your algorithm.

Solution: initalizing the array takes $O(\sum_1^n n)$ runtime, and the double for loop has the same runtime of $O(\sum_1^n n)$ b/c everything inside is O(1). Thus total runtime would be $O(\sum_1^n n)$.

Problem 4: Two Stonks *(25 points)*

Consider two possible investments A and B; each year, an oracle tells you the return of the investments for the each of the n next years. This can be represented by arrays $A = [a_1, \ldots, a_n]$ and $B = [b_1, \ldots, b_n]$ (you can assume that these are always correct). At the beginning of each year i, you can place all your investment money entirely in A or entirely in B. It is guaranteed that your money should multiply by a_i (if you invested in A) or by b_i (if you invested in B) by the end of year i. However, if you move your money from A to B or B to A when transitioning to the next year, you lose half your money.

Given the return on investments for the next n years as arrays of multipliers $A = [a_1, \ldots, a_n]$ and $B = [b_1, \ldots, b_n]$, design an algorithm to determine the maximum multiplier you can achieve on your initial investment by the end of year n.

Example: Lets say that $A = [4, 2, 3, 1]$ and $B = [1, 2, 3, 3]$. You should place your money in investment A in years 1, 2, and 3 (to have your initial investment multiply by $4 \cdot 2 \cdot 3 = 24$). Then, you would transfer your money from A to B, which would cut your money in half (now you only have a 12x multiplier on your initial investment). By the end of year 4, you would achieve a 36x multipler on your initial investment. Your algorithm should return 36 in this case.

Hint: a $1 \times n$ table will not be sufficient, but it doesn't have to be an $n \times n$ table!

(a) Define the entries of your table in words. E.g. $T[i]$ or $T[i, j]$ is ...

Solution: We use a single 2D array dp of size [n][2].dp[i][0] represents the maximum multiplier at year i when invested in A. dp[i][1] represents the maximum multiplier at year i when invested in B.

(b) State base cases and recurrence for entries of the table in terms of smaller subproblems. Briefly explain in words why it is correct.

Solution: dp[0][0] = A[0] , dp[0][1] = B[0] dp[i][0] = max(dp[i-1][0] * A[i], dp[i-1][1] * A[i] / 2), p[i][1] = max(dp[i-1][1] * B[i], dp[i-1][0] * B[i] / 2)

(c) Write **pseudocode** for your algorithm to solve this problem.

```
n = len(A)
dp = array of size (n) * (2)
for i in range(0, n - 1):
  dp[i][0] = 0
  dp[i][1] = 0

# Base case:
dp[0][0] = A[0]
dp[0][1] = B[0]
```

```
for i in range(1, n):
  # Stay in A or switch from B to A
  dp[i][0] = max(dp[i-1][0] * A[i], dp[i-1][1] * A[i] / 2)
  # Stay in B or switch from A to B
  dp[i][1] = max(dp[i-1][1] * B[i], dp[i-1][0] * B[i] / 2)

return max(dp[n-1][0], dp[n-1][1])
```

(d) Analyze the running time of your algorithm.

Solution: initalizing the array takes O(2n) times, the for loop takes O(n) times since everything inside takes O(1). Thus total runtime is O(n).