CS 3510: Design & Analysis of Algorithms October 2nd 2024

Problem Set 4: Array and String DP

Jinseo Lee Due: October 9th 2024

- Please type your solutions using LaTeX or any other software. Handwritten solutions will not be accepted.
- Your algorithms must be in plain English & mathematical expressions, and the pseudo-code is optional. Pseudo-code, without sufficient explanation, will receive no credit.
- If we ask for a specific running time, a correct solution achieving it will receive full credit even if a faster solution exists.
- Unless a question explicitly states that no work is required to be shown, you must provide an explanation or justification for your answer

Problem 1: Tomb Raider

(30 points)

You are an experienced archaeologist exploring a series of ancient ruins. Each ruin contains a certain amount of treasure, but the only problem is that adjacent ruins are protected by magical traps. If you try to collect treasure from two neighboring ruins on the same night, the traps will activate, preventing you from taking anything.

Given an integer array T, where each element represents the amount of treasure hidden in a ruin, provide an efficient algorithm that determines the maximum amount of treasure you can collect without triggering any traps.

For example, consider the array $T = [6, 7, 1, 30, 8, 2, 4]$. The maximum amount of treasure that can be collected without triggering traps is 41. This consits of collecting the treasure valued at $7, 30$ and 4.

1. Give the dimensions of your DP table and what each cell in the table stores/represents. No justification needed.

   Solution:

   DP table has 1 x (size of the input T) dimension, each cell representing the maximum amount of treasure I can collect up to the corresponding ruin.

2. Give the base case(s). Justify your answer.

   Solution:

   Since the maximum amount of treasure I can collect on the first ruin equals to T[0], the first cell in the DP table simply represents that of array T. Maximum amount of treasure on the second ruin, on the other hand, is the maximum value of T[0] and T[1] as I can't collect treasure from the two consecutive ruins. Therefore, the base cases are: DP[0] = T[0] and DP[1] = max(T[0], T[1])

3. Give the recurrence. Justify why this recurrence correctly fills in your DP table.

   Solution:

   DP[i] = max(DP[i-1], DP[i-2] + T[i]) provides each cell an curr.max amount of treasure that can be collected.

   Since I cannot steal the treasure from both of the consecutive ruins, I need to check if I should skip the current ruin and take the value of DP[i-1], or take the treasure from the current ruin and add the value of DP[i-2] to avoid triggering the trap. Therefore, recurrence above works flawlessly as DP table constantly updates the largest amount of collectable treasure possible as it progresses.

4. How do we recover the total value that can be received? Justify your answer.

Solution:

The total value could be retrieved by accessing DP[T.length - 1]. Since it is the bottom up DP table (conquering from the smallest sub-problem), the last cell represents the maximum amount of treasure collected without triggering the traps, taking all ruins into an account. More specifically, the last cell in the DP table invariably possesses the total value; Even if collecting the $(T.length - 2)^{th}$ ruin's treasures yields the maximum amount of treasure possible, the last cell will contain the identical amount as max(DP[i-1], DP[i-2] + T[i]) returns DP[i-1].

5. Give the runtime. Justify your answer.

Solution:

Computing the base cases and the recurrence both takes constant time O(1). Iterating through the DP table, filling in the corresponding value, takes O(n) time as we loop the table only once. Finally, returning the last value from the DP table takes O(1) time, making the overall runtime of this algorithm O(n).

Problem 2: Enchanted Weaving

(30 points)

You are a magical weaver tasked with combining two enchanted threads, A and B, into a single fabric C. The process must respect the original order of the threads: the symbols from both A and B must appear in C in the same relative order as they do in their respective threads. In other words, you cannot reorder the characters within A or B, but you can interleave them.

For instance, consider the fabric "`ENCHANTEDWEAVING`", which could be a possible weaving of the threads "`ENCHANTED`" and "`WEAVING`" in various ways:

1. "`ENCHWEAVINGANTED`" (take characters from both threads in an interleaved manner)
2. "`ENCHANTEDWEAVING`" (simply append B to A)
3. "`ENCHAWEAVNTEDING`" (mix characters from both threads)

Design an efficient algorithm that checks whether a given fabric C is a valid composition (or interleaving) of the two threads A and B.

1. Give the dimensions of your DP table and what each cell in the table stores/represents. No justification needed.

   Solution:

   DP table has (m + 1) x (n + 1) dimensions, where m is the length of A, and n is that of B. DP[i][j] represents whether the first i characters from A and the first j characters from B can form the first (i + j) characters from C. The first element DP[0][0] represents the base case where no characters are used from either A or B, and all other cells are filled based on interleaving decisions.

2. Give the base case(s). Justify your answer.

   Solution:

   1. DP[0][0] = True: The combination of the empty strings from A and B can form the empty string from C.

   2. $\text{DP[i][0]} = \begin{cases} \text{true}, & \text{if C[:i] == A[:i]} \\ \text{false}, & \text{otherwise} \end{cases}$

   3. $\text{DP[0][j]} = \begin{cases} \text{true}, & \text{if C[:j] == B[:j]} \\ \text{false}, & \text{otherwise} \end{cases}$

   These base cases set the foundations for the remaining cells in the DP table, establishing the cases of forming C with only A, only B, or neither.

3. Give the recurrence. Justify why this recurrence correctly fills in your DP table.

Solution:

In order to verify the validness of AB composition, we should consider the two conditions. First, the latest element in either A or B (A[i-1] or B[j-1]) should match the corresponding element in C (C[i+j-1]) for it to be in an valid order. Second, if A[i-1] matches C[i+j-1], we know that DP[i][j] must be true only and only if we can interleave the first i-1 characters of A and j characters of B (This could be checked by looking at the DP[i-1][j] value). Similarly, if B[j-1] matches C[i+j-1], it is true only and only if we can interleave the first j-1 characters of B and i characters of A (This could be checked by looking at the DP[i][j-1] value).

Therefore, the recurrence relation is: DP[i][j] = (DP[i-1][j] and A[i-1] == C[i+j-1]) or (DP[i][j-1] and B[j-1] == C[i+j-1])

4. How do we recover whether or not Fabric C is an interleaving? Justify your answer.

Solution:

We can recover the validness of C by retrieving the value at DP[m][n]. The two conditions demonstrated above (1. check whether it can be formed by taking the last character from A or B without violating the interleaving property. 2. Consider the previous DP values) are considered throughout the entire DP table cells, including DP[m][n], where all characters are considered. Therefore, the fabric C is a valid interleaving of A and B if and only if DP[m][n] is set true.

5. Give the runtime. Justify your answer.

Solution:

Computing each entry in the DP table takes constant time. However, filling in the DP table depends on the number of rows, m, and the number of columns, n, making the overall runtime proportional to size of DP table. Since the size is (m x n) + (m + n) + 1, the runtime equals to O(m x n).

Problem 3: Crazy Commuter

(40 points)

You work for a tech company called Sahara, and your role involves a mix of working from home in Des Moines and commuting to the office in Seattle. If you spend the night in Seattle, you stay at a hotel, where the cost on the night of the i^{th} day is h_i. However, if you choose to work in Seattle, Sahara also gives a bonus b_i. To travel between Des Moines and Seattle, you must book a flight costing k dollars each time you switch locations. You can switch locations once during the morning (before work) and once during the evening (after work).

Your want to determine the most profitable schedule. You begin at home in Des Moines on the morning of the first day. You never have to go to Seattle if the most cost-effective solution involves always staying in Des Moines (i.e., receiving a profit of 0). You also never have to return to Des Moines if you don't want to if you travel to Seattle. The total profit consists of the bonuses you earn for working in Seattle minus the hotel costs and the flight costs whenever you switch between cities. **Provide an efficient algorithm that finds the maximum possible profit you could have received from the given i days (i.e. on the morning of day i+1).**

As an example, consider you are given $h = [100, 120, 90, 110]$, $b = [100, 150, 20, 340]$, and $k = 200$. For this four day schedule, the optimal schedule would be to work from home on the first three days. You then fly to Seattle in the morning of the fourth day and stay there for the night. This gives you a final profit of 30 on the morning of the fifth day.

1. Give the dimensions of your DP table and what each cell in the table stores/represents. No justification needed.

   Solution:

   DP table has (2 x n), where n is the number of days. Each cell represents the maximum possible benefit I can receive in either Des Moines or Seattle. More specifically, the first row, DP[0][i], the maximum profit when I end my day $(i+1)^{th}$ in Des Moines. The second row, DP[1][i] on the other hand, represents the maximum profit when I end my $(i+1)^{th}$ day in Seattle. (0 <= i < n)

2. Give the base case(s). Justify your answer.

   Solution: On the first day, I don't have to fly to Des Moines to end my day as I have started my morning there. Therefore the base case includes DP[0][0] = 0. Furthermore, I must fly to Seattle in order to end my day there, making the base case DP[1][0] = b[0] - h[0] - k.

3. Give the recurrence. Justify why this recurrence correctly fills in your DP table.

   Solution:

   On the day i besides the very first day, I can either stay in Des Moines or fly to Seattle

if I stayed at Des Moines the previous day. Similarly, I can either stay in Seattle or fly to Des Moines if I stayed at Seattle the previous day. Considering the 'staying' conditions, staying at Des Moines consecutively doesn't cost anything, while staying at Seattle consecutively provides me with the bonus but charges me the hotel fee. Considering the 'flying' conditions, flying to Des Moines from Seattle results in the current profit subtracted by the flight cost, while flying to Seattle from Des Moines provides me with the bonus but charges me the hotel fee and flight cost. Therefore, the max value of these conditions will fill in each cells with the max profit on day $i+1$.

Therefore the recurrence is:
DP[0][i] = max(DP[0][i-1], DP[1][i-1] - k)
DP[1][i] = max(DP[1][i-1] + b[i] - h[i], DP[0][i-1] + b[i] - h[i] - k)

4. How do we recover the maximum profit? Justify your answer.

Solution:

Even if there were the absolute maximum profit during the past days, the overall profit is determined after working all n days even working additional day(s) reduces the overall profit. Therefore, after the n days, the maximum profit is the larger value between DP[0][n-1] (Staying at Des Moines) and DP[1][n-1] (staying at Seattle).

Maximum Profit = max(DP[0][n-1], DP[1][n-1])

5. Give the runtime. Justify your answer.

Solution:

There are two computations on each day, both of which takes constant time. Therefore, the runtime is proportional to the size of the DP table as we should fill in every cell.

O(size of DP table) = O(2n) = O(n)

6. How can we recover which days you're in Seattle and which days you're in Des Moines from your DP table?

Solution:

To backtrack and recover the schedule, start from the last day by comparing the values in the DP table for Des Moines and Seattle. If you end the last day in Seattle, check whether you stayed there or flew from Des Moines by using the DP values. Similarly, if you end the day in Des Moines, check if you stayed or flew back from Seattle. Repeat this process for each day, moving backward to reconstruct the sequence of decisions (stay or fly) for the entire schedule.