Theory Assignment-3: ADA

Winter-2024

Shamik Sinha (2022468) Vansh Yadav (2022559)

Preprocessing

In the provided algorithm, prior to executing the algorithm, the DP array is initialized with zeros. This preparatory step ensures that the DP array begins with a clean state, devoid of any pre-existing values.

Algorithm Description

Assumptions Made

1. The arrays are composed of integers i.e. they are integer arrays or at the very least have numeric values (floating point or integer).
2. The array indexing follows a 0-based convention. An 'n' sized array has starting index 0 and last index at 'n'-1.

Input

The input consists of two lines:

- Two integers representing the dimensions of the marble slab. The height of the slab denoted by n and the width of the slab denoted by m.
- A 2D array or matrix representing the spot prices for various rectangle dimensions. Each element of this matrix represents the price associated with a particular rectangle dimension. The dimensions of this matrix correspond to the dimensions of the marble slab, with n rows and m columns.

Output

A single integer representing the maximum profit that can be obtained by cutting the marble slab into smaller integral pieces.

Solution Description

We decided to utilise a dynamic programming approach for maximizing profit from cutting a marble slab. We decomposed the problem into sub problems of determining the maximum profit achievable for smaller slabs from the original marble slab. A detailed description of our algorithm is as follows:

1. **Initialization**:

   - Initialized a 2D array dp of size $(n+1) \times (m+1)$ with all elements initialized to 0. This array will store the maximum total price achievable for sub-slabs of various dimensions.

     The dimensions of the array are $(n+1) \times (m+1)$ because first of all, It accommodates indexing from 1, aligning with the dimensions of the original marble slab. Second, we get to define the sub problems where any one of the parameters (height or width) are equal to zero. This helps us define our sub problem more precisely and compute the maximum total price for sub-slabs of various dimensions.

2. **Populating dp array**:

   - Iterating over each cell (i, j) in the dp array where $1 \leq i \leq n$ and $1 \leq j \leq m$.

     We iterate over the range $1 \leq i \leq n$ and $1 \leq j \leq m$ because these values represent the dimensions of sub-slabs that we are considering. We start from sub-slabs of size 1×1 and gradually increase the dimensions up to the size of the original marble slab.

     We do not consider the values where any of the indices are equal to zero, as we have already initialised the array with all zeroes, in the cases where slab height or width are zero, it would mean there is no slab! and the price will automatically be zero.

   - Setting dp$[i][j]$ equal to price$[i-1][j-1]$. This step ensures that dp$[i][j]$ holds the price of the sub-slab with dimensions $i \times j$.

     By assigning the prices of individual cells of the marble slab to the corresponding cells in the dynamic programming table, we establish the base cases for our dynamic programming approach. These base cases represent the simplest sub problems, where the sub-slab dimensions are 1×1, and the maximum profit is simply the price of the single cell. As we start cutting the slabs we compare the prices set for each smaller slab with the prices of the composite sub slabs with in the smaller slabs and correspondingly update the value in the next step.

3. **Dynamic Programming**:

   - Iterating over each cell (i, j) in the dp array where $1 \leq i \leq n$ and $1 \leq j \leq m$, again.

     Now we iterate to determine the maximum profit possible for the sub slabs of size

i by j. That is also why we dont iterate with index starting ffrom zero.

- For each cell (i, j), consider all possible ways to divide the slab into two sub-slabs:
  - Horizontal cuts: Iterate over all possible positions k (where $1 \leq k < i$) to make a horizontal cut (since the size of the cut is limited by the value of the height i of the sub slab we are considering).
  - `dp[k][j]` represents the maximum total price achievable for the sub-slab with dimensions $k \times j$.
  - `dp[i - k][j]` represents the maximum total price achievable for the remaining part of the slab after the horizontal cut, with dimensions $(i - k) \times j$.
  - By adding these two values, the algorithm calculates the total price achievable by making a horizontal cut at position k.
  - Taking the maximum of `dp[i][j]` and the calculated total price helps us determine most profitable horizontal cut among all possible positions k.
  - Vertical cuts: Iterate over all possible positions l (where $1 \leq l < j$) to make a vertical cut (since the size of the cut is limited by the value of the width j of the sub slab we are considering).
  - `dp[i][l]` represents the maximum total price achievable for the sub-slab with dimensions $i \times l$.
  - `dp[i][j - l]` represents the maximum total price achievable for the remaining part of the slab after the vertical cut, with dimensions $i \times (j - l)$.
  - By adding these two values, the algorithm calculates the total price achievable by making a vertical cut at position l.
  - Taking the maximum of `dp[i][j]` and the calculated total price helps us determine most profitable vertical cut among all possible positions l.

4. **Output**:
   - After all the sub problems have been solved, the maximum total price achievable for the original slab of dimensions $n \times m$ is stored in dp$[n][m]$.
5. **Print the Answer**:
   - Output dp$[n][m]$, which represents the maximum total price achievable for the original slab, as the final answer.

Subproblem Definition

We define the sub problem using $dp[i][j]$. Given a slab of dimensions $i \times j$, $dp[i][j]$ represents the maximum total price achievable for this sub-slab.

For each (i, j) combination where $1 \leq i \leq n$ and $1 \leq j \leq m$, $dp[i][j]$ is computed based on the maximum total price obtained by considering all possible ways to cut the original slab into smaller sub-slabs.

Therefore, the 2D vector or table dp stores solutions to subproblems of increasing sizes (sub-slabs of varying dimensions), enabling the algorithm to efficiently compute the optimal solution for the original problem (cutting the entire slab) by building upon the solutions of these smaller subproblems.

Recurrence of the Subproblem

- Case 1 : When either i or j is equal to 0.

$$\text{dp}[i][j] = 0$$

- Case 2 : When neither i nor j is equal to 0.

$$\text{dp}[i][j] = \max\left(\text{dp}[i][j], \max_{1 \leq k < i}\left(\text{dp}[k][j] + \text{dp}[i-k][j]\right), \max_{1 \leq l < j}\left(\text{dp}[i][l] + \text{dp}[i][j-l]\right)\right)$$

- Overall:

$$T(n, m) = O(nm) + \sum_{i=1}^{n}\sum_{j=1}^{m}(O(n+m))$$

1. The first term, $O(nm)$, represents the initialization of the 2D array dp.
2. The second term represents the time spent within the nested loops:
   - The outer loop iterates over the length of the slab (n).
   - The inner loop iterates over the width of the slab (m).
   - The updating of the value of dp[i][j] within these additional loops indeed takes O(1) time per iteration. Since these additional loops run i - 1 times for i and j - 1 times for j, the total time spent within these loops is O(n + m).
   - Therefore, the double summation represents the total time spent within the nested loops, which is $O(n + m)$.
3. The third term represents the recursive calls made within the function. Since the function is called recursively within itself, the time complexity for each call is $T(n, m)$.

Specific Subproblem(s) for the Actual Problem

Problem Definition

Given a marble slab of dimensions $m \times n$ and an array P representing the spot prices of all possible $x \times y$ marble rectangles, where $P[x][y]$ denotes the spot price of a $x \times y$ marble rectangle, devise an algorithm to subdivide the marble slab into integral pieces to maximize profit.

Dynamic Programming Setup:

Subproblem

- dp[i][j]: Represents the maximum profit that can be obtained by cutting an $i \times j$ marble slab.

Base Case

The base case is when either i or j is equal to 1:

- When $i = 1$, dp[1][j] represents the maximum price achievable for a one-unit high slab with width j. Its value is set directly from the price matrix as price[0][$j - 1$].
- When $j = 1$, dp[i][1] represents the maximum price achievable for a slab with height i and one-unit width. Its value is set directly from the price matrix as price[$i - 1$][0].

These assignments initialize the DP table for the smallest slabs, which forms the base cases for the dynamic programming algorithm.

Final Case

The solution to the main problem, representing the maximum profit that can be obtained by cutting the marble slab into integral pieces : dp[m][n]

Time and Space Complexity

Time Complexity Analysis

1. **Initialization of the 2D Array (dp):**
   - The code initializes a 2D array `dp` with dimensions $(n + 1) \times (m + 1)$, and initializes each element to zero.
   - This initialization takes $O(nm)$ time since it involves iterating over n rows and m columns.
2. **Nested Loops:**
   - The code then iterates through the `dp` array with two nested loops.
   - The outer loop iterates over the length of the slab (n), and the inner loop iterates over the width of the slab (m).

- Within each iteration of these nested loops, there are additional loops:
  - The first additional loop runs from 1 to $i-1$ (where i is the current value of the outer loop), considering all possible cuts horizontally.
  - The second additional loop runs from 1 to $j-1$ (where j is the current value of the inner loop), considering all possible cuts vertically.
- The time complexity within these additional loops is $O(1)$ per iteration. Overall, $O(m+n)$.

.

Combining these factors, we can express the total time complexity as:

$$T(n,m) = O(nm) + O(n \cdot m \cdot (n+m))$$

This analysis captures the time spent on initialization and the nested loops. The overall time complexity is dominated by the term $O(n \cdot m \cdot (n+m))$ due to the nested loops, where each iteration takes $O(n+m)$ time.

The worst-case time complexity of the provided code snippet is $O(n^2 \cdot m + n \cdot m^2)$. This represents the scenario where both the length n and the width m of the slab contribute to the maximum possible time taken by the algorithm.

To prove that $T(n,m) = O(mn(m+n))$ is a polynomial in $m+n$, we'll demonstrate that it can be bounded above by a polynomial function in terms of $m+n$.

$$\begin{aligned} T(n,m) &= O(mn(m+n)) \\ &= O(mnx) \end{aligned}$$

Let's define $x = m+n$. Then we have:

$$T(n,m) = O(mnx)$$

Now, let's express m and n in terms of x:

$$m = x-n, \quad n = x-m$$

Substituting these expressions into $T(n,m)$, we get:

$$
\begin{aligned}
T(n,m) &= T(x-n, x-m) \\
&= O((x-n)(x-m)x) \\
&= O((x^2 - nx - mx + mn)x) \\
&= O((x^2 - (x-n)x - (x-m)x + mn)x) \\
&= O((x^2 - (x^2 + x^2 - nx - mx) + mn)x) \\
&= O((x^2 - (m+n)x + mn)x) \\
&= O((x^3 - (m+n)x^2 + mnx)
\end{aligned}
$$

Now, we observe that x^3 is the dominating term, and the others are at most quadratic. So, we can conclude that $T(n,m)$ is bounded above by a polynomial function in terms of $m+n$. Thus, we have proven that $T(n,m) = O(mn(m+n))$ is indeed a polynomial in $(m+n)$. Therefore, the time complexity of the code is polynomial in $(m+n)$.

Space Complexity Analysis

To analyze the space complexity of the provided code, let's break it down step by step:

1. **Initialization of *dp* array**:
   - We initialize a 2D vector *dp* of size $(n+1) \times (m+1)$ with all elements initialized to 0. This requires space proportional to $n \times m$, as it's a 2D array of size $n \times m$.
   - Space Complexity: $O(n \cdot m)$
2. **Updating *dp* array**:
   - The nested loops iterate over the dimensions of the *dp* array, filling it with values computed from the *price* vector.
   - We are not using any additional space here, just modifying the existing *dp* array.
   - Space Complexity: $O(1)$

Space complexity analysis:

- The dominant term in space complexity is the space required to store the *dp* array, which is $O(n \cdot m)$.

Thus, the overall space complexity of the provided code is $O(n \cdot m)$.

Summary

- **Time Complexity**: $O(mn(m+n))$
- **Space Complexity**: $O(n \cdot m)$.

Pseudocode

Below is the pseudocode which represents the logic of the C++ code we had written for the problem, in a more abstract and readable form. The function takes as input the height, width and a 2D array for the spot price of the slab. The function returns the maximum profit achievable by cutting the slab as per the information in the spot price array.

Algorithm 1 An Algorithm to find the maximum profit which can be generated by cutting a slab of height n, width m and given a corresponding 2D array "price", which has the information regarding spot prices of different slabs of smaller dimensions which might be cut from the given slab of dimension n*m.

```
1: procedure cutSlab(n, m, price)
2:     dp ← 2D array of size (n + 1) × (m + 1) filled with 0s
3:     // Populate dp with prices from the price matrix
4:     for i ← 1 to n do
5:         for j ← 1 to m do
6:             dp[i][j] ← price[i − 1][j − 1]
7:         end for
8:     end for
9:     // Dynamic programming to find maximum total price
10:    for i ← 1 to n do
11:        for j ← 1 to m do
12:            // Horizontal cuts
13:            for k ← 1 to i − 1 do
14:                dp[i][j] ← max(dp[i][j], dp[k][j] + dp[i − k][j])
15:            end for
16:            // Vertical cuts
17:            for l ← 1 to j − 1 do
18:                dp[i][j] ← max(dp[i][j], dp[i][l] + dp[i][j − l])
19:            end for
20:        end for
21:    end for
22:    // Output the maximum total price
23:    output dp[n][m]
24: end procedure
```

Proof of Correctness

Optimal Substructure

The optimal substructure property states that an optimal solution to a problem contains within it optimal solutions to subproblems. In this problem, the optimal way to cut a marble slab of size $n \times m$ can be found by considering the optimal ways to cut smaller sub-slabs within it.

Overlapping Subproblems

Overlapping subproblems arise when the solution to a problem involves solving the same subproblem multiple times. In this algorithm, the DP table dp stores the solutions to subproblems, ensuring that if the algorithm encounters the same subproblem again, it can directly access its solution from the table rather than recomputing it. This eliminates redundant computations and improves efficiency.

Proof by Induction

Base Case

When n or m is 0, the marble slab cannot be cut, so the profit is 0. This base case is correctly handled in the initialization step where $dp[i][j]$ is set to 0 for all i and j where i or j is 0.

Inductive Step

Assume that the algorithm correctly computes the maximum profit for all marble slabs smaller than the current size. We need to show that it also computes the maximum profit for the current size marble slab.

The algorithm considers all possible cuts (horizontal and vertical) for the current marble slab. By considering all possible cuts and maximizing the profit at each step, it ensures that the optimal solution is achieved.

Termination

The algorithm terminates when it fills the entire DP table dp. This happens after considering all possible combinations of cuts for each sub-slab, ensuring that the maximum profit for the entire marble slab is computed.

Conclusion

The proof by induction establishes that the algorithm correctly computes the maximum profit achievable by cutting the marble slab into integral pieces. The optimal substructure and overlapping subproblems properties ensure that the algorithm's approach is valid and efficient, leading to the correct solution.