Theory Assignment-3: ADA Winter-2024

Aarzoo (2022008) Anushka Srivastava (2022086)

1 Subproblem Definition

The subproblem can be defined as $dp[i][j]$ where i and j are the height and width of the marble slab, and $dp[i][j]$ represents the maximum profit which can be achieved in a marble slab of $i \times j$ dimensions.

2 Recurrence of the sub-problem

The subproblem can be summarised as below:

$$dp[n][m] = \max \left\{ \begin{array}{l} \text{cost}(n, m), \\ \max_{1 \le i \le \lfloor n/2 \rfloor} \{dp[i, m] + dp[n - i, m]\}, \\ \max_{1 \le j \le \lfloor m/2 \rfloor} \{dp[n, j] + dp[n, m - j]\} \end{array} \right\}$$

The explanation of this subproblem is given in detail in the **Algorithm Description** section.
The recurrence relation which solves the problem is:

$$T(n, m) = \sum_{i=1}^{\frac{n}{2}} (T(i, m) + T(n - i, m)) + \sum_{j=1}^{\frac{m}{2}} (T(n, j) + T(n, m - j)) + c$$

3 Complexity Analysis

- The algorithm is based on the above recurrence relation. In the above algorithm, there are a total of $n \times m$ values in the dp array, and in the worst case, a total of $m + n$ operations are performed to compute the maximum value to be stored in the dp array, which happens in the case of calculating $dp[n][m]$. Thus, the time complexity is O($n \times m$)* O($m+n$) which implies the overall time complexity would be $O(n*m(n+m))$.
- The depth of the recurrence relation in the worst case is m+n. Consider a case where either n = 1 or m = 1, more specifically, only either horizontal cuts are possible or vertical cuts are possible. For instance, if only vertical cuts are possible (n = 1), then the cost for m + 1 slabs need to be evaluated to compute the result of dp[1][m], Similiarily , when m = 1 , cost of n + 1 slabs need to be evaluated to compute the result of dp[n][1] . The same holds true also for m and n not equal to 1. Therefore, in the worst case the time complexity is $O(n * m(n + m))$ where $n \times m$ represents the values and $n + m$ represents the operations performed.
- The space complexity of the algorithm is $O(n * m)$, since the dp array is initialized with the size of $(n + 1) \times (m + 1)$.

4 Specific subproblem that solves the actual problem

The final answer is obtained by returning the value of $dp[n][m]$ where n and m are the height and width of the marble slab given as input.

5 Algorithm Description

Our algorithm uses dynamic programming using memoization to solve this problem. The dimensions of our marble are given as n centimetres in height and m centimetres in width which gives us the dimensions as $n \times m$. The cost matrix is also given as input with dimensions $(n+1) \times (m+1)$ where (i, j) represents the cost of the marble slab of dimension $i \times j$. Our algorithm follows 1-based indexing.

5.1 Initialization of dp matrix:

We initialize a 2D array of dimensions $(n+1) \times (m+1)$ as a dp matrix. In this matrix, $dp[i][j]$ represents the maximum profit a marble slab of $i \times j$ dimension can return.

5.2 Recursive function (Memoization):

Base Cases:

- When $m = 0$ or $n = 0$: This indicates that the slab cannot be divided further. Hence, we store 0 in $dp[n][m]$. This recursive base step is added to maintain the integrity of 1-based indexing.
- When $dp[n][m] \neq -1$: This indicates that the maximum profit for $n \times m$ slab has already been calculated. So, there is no need for further computation, and the value of $dp[n][m]$ is simply returned. This step highlights the significance of the memoization algorithm.

Recursive steps:

In the rest of the function, we use our subproblem definition to do further calculations. According to our subproblem, to calculate the maximum profit from a slab of $n \times m$ dimension, we have to calculate the maximum value out of three possible combinations: when the slab is not cut, when there is a horizontal cut at ith centimeter from the top and when there is a vertical cut at jth centimeter from the left. Here, n and m are the height and width of the marble slab in that particular recursive stack.

- **Horizontal cut:**

  We only iterate from $i = 1$ to $i = n/2$ due to the principle of symmetry. By making a cut at ith centimeter from the top, we get two slabs of the size $i \times m$ and $(n-i) \times m$. The maximum profit for these two slabs is calculated by calling the recursive function on these parameters again until either one of the base cases is hit or the slab cannot be divided further (that is, a 1×1 slab).

- **Vertical cut:**

  We only iterate from $j = 1$ to $j = m/2$ due to the principle of symmetry. By making a cut at jth centimeter from the left, we get two slabs of the size $n \times j$ and $n \times (m-j)$. The maximum profit for these two slabs is calculated by calling the recursive function on these parameters again until either one of the base cases is hit or the slab cannot be divided further (that is, a 1×1 slab).

- **No cut:**

  We simply return $cost[n][m]$, where this is the cost of the marble slab of $n \times m$ dimension, indicating that there was no cut.

  We have 3 values of all the possible cases at a particular step. Since we need to calculate the maximum profit we can get, we compute the maximum of these values and store it at $dp[n][m]$.

After the recursion ends, we have the dp array filled with $dp[i][j]$ filled with the maximum profit achievable from $i \times j$ slab.

The final answer is obtained by returning the value of $dp[n][m]$ where n and m are the height and width of the marble slab given as input.

6 Reference

- **Math StackExchange**

7 Pseudocode

Algorithm 1 Maximum Profit Calculation

```
function MaximumProfit(cost, dp, n, m)
    if m == 0 or n == 0 then
        dp[n][m] ← 0
        return 0
    end if
    if dp[n][m] ≠ −1 then
        return dp[n][m]
    end if
    maxProfit ← 0
    for i ← 1 to n/2 do
        m1 ← MaximumProfit(cost, dp, i, m)
        m2 ← MaximumProfit(cost, dp, n − i, m)
        maxProfit ← max(maxProfit, m1 + m2)
    end for
    for j ← 1 to m/2 do
        m1 ← MaximumProfit(cost, dp, n, j)
        m2 ← MaximumProfit(cost, dp, n, m − j)
        maxProfit ← max(maxProfit, m1 + m2)
    end for
    maxProfit ← max(maxProfit, cost[n][m])
    dp[n][m] ← maxProfit
    return dp[n][m]
end function

function CalculateProfit(n, m, cost)
    Initialize dp with size (n + 1) × (m + 1), filled with −1
    max_prof ← MaximumProfit(cost, dp, n, m)
    return max_prof
end function
```