Theory Assignment-3: ADA Winter-2024

Himang Chandra Garg (2022214) Nishil Agarwal (2022334)

1 Preprocessing

Not Applicable in this Algorithm.

2 Assumption

One cannot rotate a block of marble. Let's say the cost of the (i,j) size block is x, and for (j, i) size it is y. Then, a block of size height = i, width = j is sold for x only. It cannot be sold for y by rotating.

3 Subproblem

We calculate this value at each step:

dp[h][i] = The maximum profit that can be obtained by selling (h+1)*(i+1) centimeters of the slab.

Here, h+1 represents the height of the current marble slab, and i+1 represents the width of the current marble slab.
We have started counting h and i from 0 to m-1 and n-1, respectively. The actual height and width of a slab of dimensions (h, i) is (h+1,i+1).

4 Reccurrence of Subproblem

$$dp[h][i] = \max \left\{ \begin{array}{l} v[h][i] \\ dp[k][i] + dp[h-k-1][i] \\ dp[h][j] + dp[h][i-j-1] \end{array} \right\}$$

Here, j is varying from 0 to i-1. k is varying from 0 to h-1.
Hence, $dp[k][i] + dp[h-k-1][i]$ represents h different terms. Similarly, $dp[h][j] + dp[h][i-j-1]$ represents i different terms.
We are taking the maximum of all of these terms combined together.

5 Subproblem that solves the final problem

$$\max \left\{ \begin{array}{l} v[m-1][n-1] \\ dp[k][n-1] + dp[m-k-2][n-1] \\ dp[m-1][j] + dp[m-1][n-j-2] \end{array} \right\}$$

Here, j is varying from 0 to n-1. k is varying from 0 to m-1.
Hence, $dp[k][i] + dp[h-k-1][i]$ represents m different terms. Similarly, $dp[h][j] + dp[h][i-j-1]$ represents n different terms.
We are taking the maximum of all of these terms combined together.

6 Algorithm Description

We have used tabularization or a bottom-up approach of Dynamic Programming using a 2-D array.

Here, dp[h][i] stores the maximum price that we can get by selling a slab of size (h+1)*(i+1).

We loop through m height and n width to access all cells of the dp table. At each iteration, we populate dp[h][i] with the maximum money gainable from selling marble of size (h+1)*(i+1).

We calculate the maximum price of (h+1)*(i+1) size by the following cases:

1. Either the spot price for (h+1)*(i+1) size slab as a whole is the best price we can get by selling the slab.
2. Or we can sell it in same way as we would sell a (k+1)*(i+1) and a ((h+1)-(k+1))*(i+1) size slab. We would be selling the same (h+1)*(i+1) slab but in a different way. Here k will vary from 0 to h-1 to cover all cases.
3. Or we can sell it in same way as we would sell a (h+1)*(j+1) and a (h+1)*((i+1)-(j+1)) size slab. We would be selling the same (h+1)*(i+1) slab but in a different way. Here j will vary from 0 to k-1 to cover all cases.

All of these possible cases are taken into account for the calculation of the max price for [h][i]$^{\text{th}}$ cell of dp table. We calculate the max out of these h+i+1 options for this.

We will continue populating the dp till [m-1][n-1]$^{\text{th}}$ cell representing the max sell price of slab of dimensions (m)*(n). This will be our answer.

7 Runtime Complexity Analysis

The time complexity of this Algorithm is:

$$O(mn(m+n))$$

Here, m is the height of the slab, and n is the width of the slab initially given to us.

We came to this result since our code contains nested operations:
1. Topmost for loop that traverses h from 0 to m-1(m operations).
2. Second level for loop that traverses i from 0 to n-1(n operations).
3. There are 2 bottom-level loops that generate i+h cases for calculating the maximum selling price. Also, the max function used to find the maximum price out of these i+h+1 cases has a time complexity of O(n+m).

Since the operations are nested, the time complexity gets multiplied.
The resulting time complexity in terms of (m+n) can also be written as:

$$O((m+n)^3)$$

Since Big O gives only the upper bound.

8 Pseudocode

Algorithm 1 Maximize Profit Algorithm

```
procedure MaximizeProfit(m, n, v)                ▷ v is a 2D array representing spot prices
    dp ← Initialize a 2D array of size m × n with all elements set to −1
    for h ← 0 to m − 1 do
        for i ← 0 to n − 1 do
            temp ← Empty list
            Append v[h][i] to temp
            for k ← 0 to h − 1 do
                Append dp[k][i] + dp[h − k − 1][i] to temp
            end for
            for j ← 0 to i − 1 do
                Append dp[h][j] + dp[h][i − j − 1] to temp
            end for
            dp[h][i] ← Maximum value in temp
        end for
    end for
    Output dp[m − 1][n − 1]
end procedure
```