Theory Assignment-3: ADA Winter-2024

Himanshu Raj (2022216)   Tanish Verma (2022532)

February 20, 2024

1 Preprocessing

Given m, n and a 2D array P, construct a two-dimensional dp array of size m x n. Initial values of dp array don't matter as we will update the entire array during the execution of our algorithm.
We will follow 0-based indexing.

2 Subproblem Definition

dp[i][j] →The maximum profit from a marble of size (i+1) x (j+1), according the spot prices in array P.

3 Recurrence Relation

Base Cases
There is no particular base case as such, since the recurrence relation itself takes care of the base case. This is because (refer recurrence below), for cases where such base conditions may arise, like P[0][0], the recurrence itself takes care of it such that they directly take the values from P array itself (since, the for loops won't be executed).

Recurrence of the Subproblem

```
for (i = 0 to m - 1):
    for (j = 0 to n - 1):
```

$$dp[i][j] = max \left\{ \begin{array}{l} P[i][j] \\ for(k = 0\, to\, i-1): \\ \qquad dp[i-k-1][j] + dp[k][j] \\ for(k = 0\, to\, j-1): \\ \qquad dp[i][j-k-1] + dp[i][k] \end{array} \right\}$$

Note: for (k = 0 to i - 1) means that both initial and final indexes are inclusive.

4 Subproblem that solves the actual problem

dp[m-1][n-1] → returns the maximum profit from the given marble of size mxn.

5 Algorithm Description

The algorithm iterates over all cells of the rectangle, considering each cell as the bottom-right corner of a subrectangle. For each cell, it computes the maximum cost achievable for the corresponding subrectangle.

This is done by two nested loops

```
for (i = 0 to m - 1):
    for (j = 0 to n - 1):
        and for each entry, we apply the recurrence relation mentioned above.
```

The Recurrence relation can be explained as, for a given subproblem, i.e, dp[i][j], there are three possible cases to compute maximum profit:

1. The cost of (i,j) partition itself, i.e., i x j slab.
2. For each possible horizontal partition, the algorithm computes the cost of both slabs in the partition and selects the maximum among them. It iterates through all possible partition heights 'k' within the range `[0, i - 1]` and calculates the maximum cost considering the partition at height `k`. The maximum cost of the two partitions here is computed by checking the dp array corresponding to that partition.
3. For each possible vertical partition, the algorithm computes the cost of both slabs in the partition and selects the maximum among them. It iterates through all possible partition widths 'k' within the range `[0, j - 1]` and calculates the maximum cost considering the partition at width `k`. The maximum cost of the two partitions here is computed by checking the dp array corresponding to that partition.

Finally, the maximum of all these three cases is taken and stored in the dp array for each (i,j) index.

Once all cells are processed, the maximum cost achievable for the entire rectangle is obtained from the bottom-right corner of the rectangle, which is stored in `dp[m-1][n-1]`.

6 Pseudocode

```
function MaxProfit(m, n, P, dp)
    for (h : 0 to m-1):
        for (w : 0 to n-1):

            a = P[h][w]
            b = -∞
            c = -∞

            for(k : 0 to h-1):
                b = max (b, dp[h-k-1][w] + dp[k][w])
            for(k : 0 to w-1):
                c = max (b, dp[h][w-k-1] + dp[h][k])

            dp[h][w] = max(a,b,c)

    return dp[m-1][n-1]
end function
```

7 Complexity Analysis

7.1 Time complexity

During the algorithm, we are iterating over the array P once, which are m*n iterations. In each iteration, we perform some constant time operations and iterate over a dp row and column to check for maximum from all possible horizontal and vertical cuts which costs us O(m+n) time at worst (due to two consecutive for-loops inside each iteration). So running time complexity of our algorithm is O(m*n*(m+n)).

7.2 Space complexity

We are using an additional dp array of size m x n, so additional space complexity of our algorithm is O(m*n).