# Theory Assignment-3: ADA Winter-2024

Nikhil Kumar (2022322) Nikhil (2022321)

20 February 2024

## 1 Subproblem Definition

The subproblem addressed in the provided code involves finding the maximum profit achievable by cutting a marble slab of dimensions $(m \times n)$ into smaller pieces. The algorithm systematically explores all possible subproblems, with each subproblem corresponding to a smaller section of the original slab.

The subproblem can be formally defined as follows:

Given a marble slab of dimensions $(m \times n)$ and a matrix $(P)$ representing the spot prices of each piece, determine the maximum profit that can be obtained by strategically cutting the slab into smaller pieces.

## 2 Recurrence of the subproblem

For $i$ from 1 to $m$ and $j$ from 1 to $n$, the recurrence relation is defined as follows:

The recurrence relations for the given code can be expressed as follows:

$$DP[i][j] = A$$

$$\begin{aligned}
\text{Vertical case:} \quad & DP[i][j] = A \\
& A = \max(DP[i][j], DP[i][\text{vertical\_cut}] + DP[i][j - \text{vertical\_cut}]) \\
& \text{for vertical\_cut from 1 to } j \\
\text{Horizontal case:} \quad & DP[i][j] = A \\
& A = \max(DP[i][j], DP[\text{horizontal\_cut}][j] + DP[i - \text{horizontal\_cut}][j]) \\
& \text{for horizontal\_cut from 1 to } i
\end{aligned}$$

Where:

- $I$ ranges from 1 to $m$
- $J$ ranges from 1 to $n$
- `Horizontal_cut` ranges from 1 to $i$
- `Vertical_cut` ranges from 1 to $j$

## 3 The specific subproblem(s) that solves the actual problem

The specific subproblem being solved in this code is to find the maximum profit that can be obtained by cutting a rectangle of size $(i \times j)$ into smaller rectangles. Each cell in the array $P$ represents a rectangle with a certain price. The function `maximizeProfit` utilizes dynamic programming to compute this maximum profit.

Specifically, it iterates through all possible rectangle sizes up to $(m \times n)$, filling in a DP table $DP[i][j]$ to store the maximum profit for each subproblem of size $(i \times j)$. For each rectangle size $(i \times j)$, it considers all possible horizontal and vertical cuts to divide the rectangle into smaller rectangles and calculates the maximum profit based on the optimal cuts.

## 4 Algorithm Description

To solve the problem, we use dynamic programming with tabulation. We define the DP table as a 2D matrix of size $(m + 1) \times (n + 1)$, so that we can handle the cases when both the height and width are zero. We initialize the 2D matrix table with 0 as the initial value at each index.

The main idea of the algorithm is to consider all the possible cases to cut the rectangle both horizontally and vertically, and then choose the maximum of them. We fill the DP table iteratively and return the final answer as the value at the index $(m, n)$ of the DP table.

1. We start with a loop from 1 to $m$, both inclusive, to consider all the possible widths. Let $i$ be the loop variable.
2. For every $i$ of the outer loop (considering all possible widths), we use a nested loop from 1 to $n$, both inclusive, to consider all the possible heights. Let $j$ be the loop variable. For every pair $(i, j)$, we have a pair of (width, height).
3. Initially, we set the price for the pair $(i, j)$ as the price without any cuts, i.e., the price indicated by the given 2D matrix of prices.
4. For every pair $(i, j)$, we first consider vertical cuts by iterating from 1 to $j - 1$. Let the variable be $vertical\_cut$. If we cut the current rectangle of $(i, j)$ into a smaller rectangle of size $(i, vertical\_cut)$, then the other part of the rectangle would be $(i, j - vertical\_cut)$.

   We calculate the profit as the profit for $(i, vertical\_cut) + (i, j - vertical\_cut)$. We access these values from the given 2D price matrix in $O(1)$ time. Finally, we store the maximum of the original price (price for $(i, j)$) and the new price we calculated at the index $(i, j)$ of the DP table.
5. Similarly, we consider the horizontal cuts by iterating from 1 to $i - 1$. Let the variable be $horizontal\_cut$. If we cut the current rectangle of $(i, j)$ into a smaller rectangle of size $(horizontal\_cut, j)$, then the other part of the rectangle would be $(i - horizontal\_cut, j)$.

   We calculate the profit as the profit for $(horizontal\_cut, j) + (i - horizontal\_cut, j)$. We access these values from the given 2D price matrix in $O(1)$ time. Finally, we store the maximum of the original price (price for $(i, j)$) and the new price we calculated at the index $(i, j)$ of the DP table.
6. Repeating the above steps for every pair of $(\text{width}, \text{height})$ will give us the maximum profit, which is obtained at the index $(m, n)$ of the DP table.

## 5 Complexity Analysis

1. Initialization of DP Array: Initialize a 2D DP array of size $(m+1) \times (n+1)$, which takes $O(m \times n)$ time in creation.

2. Nested Loops: There are two nested loops iterating over all possible subproblems.

   - The outer loop runs from 1 to $m$.
   - The inner loop runs from 1 to $n$.

   Therefore, the time complexity of these nested loops is $O(m \times n)$.

3. Vertical Cuts Loop: Similarly, inside the nested loops, there's another loop iterating from 1 to $j-1$. This loop is responsible for considering all possible vertical cuts. In the worst case, it goes up to $j-1$, where $j$ is the height of the marble slab.
   Therefore, the time complexity of this loop is $O(j)$, where $j$ can range from 1 to $n$ in the worst case. Hence the time complexity here would be: $O(n)$.

4. Horizontal Cuts Loop: Inside the nested loops, there's another loop iterating from 1 to $i-1$. This loop is responsible for considering all possible horizontal cuts. In the worst case, it goes up to $i-1$, where $i$ is the width of the marble slab.
   Therefore, the time complexity of this loop is $O(i)$, where $i$ can range from 1 to $m$ in the worst case. Hence the time complexity here would be: $O(m)$.

5. Max Operation: Inside both the horizontal and vertical cuts loops, there's a `max` operation, which does not affect the overall time complexity as it takes constant time.

Therefore, the overall time complexity of the code is $O(m \times n \times (m+n))$.

## Space Complexity Analysis

The space complexity for this algorithm is determined by the space required for the DP table, which is a 2D array storing the values of the subproblems. The size of the DP table is $O((m+1) \times (n+1))$.

Therefore, the space complexity of this algorithm is $O((m+1) \times (n+1))$.

## 6 Pseudocode

```
function MaximizeProfit($P, m, n$)
    Initialize a 2D array $DP$ of size $(m+1) \times (n+1)$ with zeros
    for $i \leftarrow 1$ to $m$ do
        for $j \leftarrow 1$ to $n$ do
            $DP[i][j] \leftarrow P[i-1][j-1]$
            for vertical_cut $\leftarrow 1$ to $j-1$ do
                $DP[i][j] \leftarrow \max(DP[i][j], DP[i][\text{vertical\_cut}] + DP[i][j - \text{vertical\_cut}])$
            end for
            for horizontal_cut $\leftarrow 1$ to $i-1$ do
                $DP[i][j] \leftarrow \max(DP[i][j], DP[\text{horizontal\_cut}][j] + DP[i - \text{horizontal\_cut}][j])$
            end for
        end for
    end for
    return $DP[m][n]$
end function
```