Theory Assignment-3: ADA Winter-2024

Devansh Grover (2022151) Garvit Kochar (2022185)

1 Subproblem Definition

We are given a marble slab of size $m \times n$ and a 2D array P of spot prices for marble rectangles of all possible sizes. Our goal is to cut the slab into smaller rectangles to maximize the profit.
The key idea in the solution is to consider all possible ways of cutting the slab and take the one that gives the maximum profit. This is where the concept of subproblems comes in. A subproblem in this context is the problem of finding the maximum profit that can be obtained by cutting a smaller slab of size $i \times j$.

Let there be a function $maxProfit(i, j)$ that computes the maximum profit for a slab of size $i \times j$. It does this by considering all possible cuts:

- The slab is sold as is for a profit of $P[i-1][j-1]$.
- $H(i, j)$ represents the maximum profit obtained by making a horizontal cut at every possible position k and recursively solving the subproblems for the two resulting slabs of sizes $k \times j$ and $(i-k) \times j$.
- $V(i, j)$ represents the maximum profit obtained by making a vertical cut at every possible position l and recursively solving the subproblems for the two resulting slabs of sizes $i \times l$ and $i \times (j-l)$.

We observe that the maximum profit for the slab of size $i \times j$ is then the maximum of the profits obtained from these three options.

2 Recurrence of the subproblem

Base Case: The base case occurs when the dimensions of the $i \times j$ marble slab are non-positive, i.e., when either i or j is less than or equal to 0. In such cases, the slab cannot be subdivided further and the profit is 0. That is,

$$\text{maxProfit}(i, j) = 0 \text{ if } i \leq 0 \text{ or } j \leq 0$$

We form 3 cases:
1. The slab is sold as is, i.e, it is sold for the spot price P[i-1][j-1].

2. $H(i, j)$**:** This means that for each possible horizontal cut at position k (where $1 \leq k < i$), we divide the slab into two smaller slabs of sizes $k \times j$ and $(i-k) \times j$. The maximum profit for this cut is the sum of the maximum profits of these two smaller slabs, i.e., $maxProfit(k, j) + maxProfit(i-k, j)$. We take the maximum over all possible k, which gives us the maximum profit for horizontal cuts, $H(i, j)$.

$$H(i, j) = \max_{1 \leq k < i}(maxProfit(k, j) + maxProfit(i-k, j))$$

3. $V(i, j)$**:** This means that for each possible vertical cut at position l (where $1 \leq l < j$), we divide the slab into two smaller slabs of sizes $i \times l$ and $i \times (j-l)$. The maximum profit for this cut is the sum of the maximum profits of these two smaller slabs, i.e., $maxProfit(i, l) + maxProfit(i, j-l)$. We take the maximum over all possible l, which gives us the maximum profit for vertical cuts, $V(i, j)$.

$$V(i, j) = \max_{1 \leq l < j}(maxProfit(i, l) + maxProfit(i, j-l))$$

The maximum profit for the slab of size $i \times j$ is the maximum of the profits obtained from these three options. $maxProfit(i, j)$ is thus defined as:

$$\text{maxProfit}(i, j) = \max\{P[i-1][j-1], H(i, j), V(i, j)\}$$

3 Specific Subproblem(s) that solve the actual Problem

Our actual problem is to find the maximum profit that is achievable when we are given a $m \times n$ rectangular slab and we cut it into smaller rectangles. This is solved by calling $maxProfit(m, n)$, which has the recurrence of the form:

$$\text{maxProfit}(m, n) = \begin{cases} 0 & \text{if } m \leq 0 \text{ or } n \leq 0 \\ \max\{P[m-1][n-1], H(m, n), V(m, n)\} & \text{otherwise} \end{cases}$$

4 Algorithm Description

1. Pre-processing: Create a 2D array/vector *dp* of size $(m+1) \times (n+1)$. This will act as our memoization table, being used to store the maximum profit that can be obtained by cutting a marble slab of size $i \times j$. Initialize all the values in this table to -1.

2. Function *maxProfit*: This is a recursive function that takes the dimensions of the slab *m, n*, 2D spot price array P and the *dp* array as input parameters.

3. Base Case: If either m or n is less than or equal to 0, the function returns 0 as there is no rectangle to be cut.

3. Memoization: If the maximum profit for the current rectangle size has already been computed (i.e., *dp*$[m][n]$ is not -1), the function returns the stored value.

4. Compute Profit Without Cut: The algorithm then computes the profit that can be obtained by selling the entire marble slab of size $m \times n$ without any cuts, which is $P[m-1][n-1]$. This value is obtained in O(1) time as per the question, and a variable *profit* is intialized with it.

5. Compute Profit with Horizontal Cuts: The function then iterates over all possible horizontal cuts (from $i = 1tom - 1$). For each cut, it recursively computes the maximum profit for the two resulting rectangles ($i \times n$ and $(m-i) \times n$) and updates $profit$ if the sum of these profits is greater than the current $profit$.

6. Compute Profit with Vertical Cuts: Similarly, the function iterates over all possible vertical cuts (from $j = 1ton - 1$). For each cut, it recursively computes the maximum profit for the two resulting rectangles ($m \times j$ and $m \times (n-j)$) and updates $profit$ if the sum of these profits is greater than the current $profit$.

7. Memoization Update: The function stores the computed maximum profit for the current rectangle size in *dp*$[m][n]$.

8. Return Value: Finally, the function returns the maximum profit for the current rectangle size by returning the value of $profit$.

This algorithm uses a top-down dynamic programming approach with memoization to avoid redundant computations. The key idea is to try all possible cuts and choose the one that yields the maximum profit. The use of a cache allows the algorithm to reuse previously computed results, significantly speeding up the computation. As mentioned earlier, the algorithm assumes that the spot price P[x, y] for any $x \times y$ marble rectangle can be queried in O(1) time (as given in the question).

Algorithm 1 Pseudocode for maxProfit

```
function maxProfit($(m, n, P, dp)$)
    if $m \leq 0$ or $n \leq 0$ then
        return 0
    end if
    if $dp[m][n] \neq -1$ then
        return $dp[m][n]$
    end if
    profit $\leftarrow P[m-1][n-1]$
    for $i \leftarrow 1$ to $m-1$ do
        profit $\leftarrow$ max(profit, maxProfit$(i, n, P, dp)$ + maxProfit$(m-i, n, P, dp)$)
    end for
    for $j \leftarrow 1$ to $n-1$ do
        profit $\leftarrow$ max(profit, maxProfit$(m, j, P, dp)$ + maxProfit$(m, n-j, P, dp)$)
    end for
    $dp[m][n] \leftarrow$ profit
    return profit
end function
```

Note: We are using zero based indexing. To traverse the input Price matrix, we start each **for** loop from 1. Also note that in each **for** loop, each index is inclusive (i.e., we are using $\leq$ instead of $<$).

5 Running time of the Algorithm

The running time of the $maxProfit$ function is $O(m^2n + mn^2)$.
We can observe that $m^2n + mn^2$ is a polynomial in terms of m and n, with the highest degree being 2.

Explanation:
The time complexity of the algorithm is determined by the number of unique states in the recursion and the work done for each state.
The dp array is used to store the maximum profit that can be obtained from a slab of size $m \times n$. If the value of $dp[m][n]$ has already been computed, it's returned immediately in O(1) time. This avoids redundant computation and is the key to reducing the running time.

In each recursive call, we explore all possible partitions in both dimensions, which means the total number of recursive calls is proportional to the number of ways we can partition the rectangular slab in both dimensions. In this case, the unique states (possible paritions) are determined by the variables m and n, which can take values from 1 to m and 1 to n respectively. Therefore, there are $m * n$ unique states. Inside each function call, the function performs two loops: one iterating i from 1 to m, and another iterating j from 1 to n. In both these loops, $maxProfit$ is called again on a smaller input. While returning, since the result of each of these function calls is stored in the dp table, each call is only computed once. Therefore, the work done for each state is the time complexity of the **for** loops, which is $O(m + n)$. Multiplying the number of unique states by the work done per state, the overall time complexity of the function is $O(m^2n + mn^2)$.