Theory Assignment-3: ADA Winter-2024

Shobhit Raj (2022482) Vashu (2022606)

1 Subproblem Definition

MaxProfit(i_1, j_1, i_2, j_2) represents the maximum profit that can be obtained by subdividing a marble slab of dimensions $j_1 \times j_2$ into integral pieces, where i_1 and j_1 represent the starting pointers of the width and i_2 and j_2 represent the starting and ending pointer of the height.

2 Recurrence relation for the subproblem

$$\begin{aligned}\text{MaxProfit}(i_1, j_1, i_2, j_2) = \max \Bigg\{ & \text{price}[n][m], \\ & \left(\max_{\text{ind}_1 \in (i_1 \text{ to } j_1)} \{\text{MaxProfit}(i_1, \text{ind}_1, i_2, j_2) + \text{MaxProfit}(\text{ind}_1, j_1, i_2, j_2)\} \right), \\ & \left(\max_{\text{ind}_2 \in (i_2 \text{ to } j_2)} \{\text{MaxProfit}(i_1, j_1, i_2, \text{ind}_2) + \text{MaxProfit}(i_1, j_1, \text{ind}_2, j_2)\} \right) \Bigg\}\end{aligned}$$

3 Specific Subproblem that Solves the Final Problem

MaxProfit$(1, n, 1, m)$ represents the maximum profit when dividing the slab of size $n \times m$.

4 Algorithm Description

4.1 Pseudocode

Algorithm 1 Maximum Profit Algorithm

Input: Size of the marble slab m and n, and $price[][]$ array
Output: Maximum profit obtained by dividing the marble slab of size $m \times n$
Assumption:1-based indexing and $h \leq n$ and $w \leq m$
Initialization:

```
Initialize a 2-D table dp[][] with dimensions n × m
for i ← 0 to n do
    for j ← 0 to m do
        Initialize dp[i][j] ← −1
    end for
end for
function MaxProfit(i1, j1, i2, j2)
    if i1 > j1 then
        return 0
    end if
    if i2 > j2 then
        return 0
    end if
    w ← j1 − i1 + 1
    h ← j2 − i2 + 1
    maximum1 ← price[h][w]
    maximum1 ← price[h][w]
    if dp[h][w] ≠ −1 then
        return dp[h][w]
    end if
    cost ← 0
    for ind1 from i1 to j1 do
        cost ← MaxProfit(i1, ind1, i2, j2) + MaxProfit(ind1 + 1, j1, i2, j2)
        maximum1 ← max(maximum1, cost)
    end for
    cost ← 0
    for ind2 from i2 to j2 do
        cost ← MaxProfit(i1, j1, i2, ind2) + MaxProfit(i1, j1, ind2 + 1, j2)
        maximum2 ← max(maximum2, cost)
    end for
    dp[h][w] ← max(maximum1, maximum2)
    return dp[h][w]
end function
```

4.2 Detailed Description

The algorithm aims to find the maximum profit that can be obtained by dividing a marble slab of size $m \times n$ into smaller integral pieces, considering the prices of each piece stored in the price$[m][n]$ array.

The algorithm employs a recursive dynamic programming approach to solve the problem. It can be categorized as a top-down dynamic programming approach, where solutions to smaller subproblems are computed first and then used to solve larger subproblems. The use of dynamic programming ensures that the algorithm avoids redundant computations and optimizes efficiency.

We first initialize the table with -1. Then, we follow a top-down approach to update the table based on the recurrence relations described earlier. Finally, we return the maximum profit achievable for the $m \times n$ slab.

Base Cases: The algorithm begins by checking base cases to handle invalid dimensions of the marble slab. If either the width or the height of the slab is zero or negative (i.e., $i1 > j1$ or $i2 > j2$), the algorithm returns 0, indicating that no profit can be obtained from such subdivisions.

Recursive Exploration: The core of the algorithm lies in its recursive function `MaxProfit(i1, j1, i2, j2)`, which explores all possible ways of partitioning the marble slab into smaller integral pieces. By recursively dividing the slab into smaller sub-slabs and considering all possible horizontal and vertical cuts, the algorithm exhaustively searches for the configuration that maximizes the total profit.

Thus, through the combination of recursive exploration, dynamic programming, and memorization techniques, the algorithm ensures correctness and efficiency in determining the optimal solution.

5 Run time analysis

Subproblems: The algorithm breaks down the original problem into smaller subproblems by considering all possible cuts horizontally and vertically. For each possible cut, it recursively computes the maximum profit of the resulting subproblems. The number of subproblems depends on the dimensions of the marble slab. Time Complexity Without Memoization is in the order of Exponential.

Memoization: The memoization technique ensures that each subproblem is solved only once, and the result is stored for future reference. This prevents redundant calculations and improves efficiency, as it can access the already computed profit of the subproblem in $O(1)$-time.

With memoization, the number of unique subproblems is $O(m \times n)$, represented by the $dp[m][n]$ array which saves the state of $m \times n$ subproblems, each unique combination. Inside each subproblem, the algorithm makes recursive calls to explore possible subdivisions. There are two for loops: one runs m times and the second runs n times, resulting in $O(m+n)$ computational effort for each recursive call. Therefore, the overall time complexity is $O(m \times n \times (m+n))$, considering the number of unique subproblems and the effort per subproblem. Thus, memoization significantly improves the efficiency of the algorithm by avoiding redundant work and reducing the time complexity to a more manageable $O(m \times n \times (m+n))$.

Overall, the runtime complexity of the provided code is dominated by the iterative bottom-up approach, which is approximately $O((M+N)^3)$.