Marble Slab Cutting Algorithm Analysis

Aditya Kumar(2022033) Nishant Kumar(2022326)

February 20, 2024

1 Algorithm Description

The `maxProfit` algorithm is designed to maximize profit by cutting a marble slab into smaller rectangles. It uses "**top to bottom**" dynamic programming to compute the maximum profit of cutting a marble slab. The algorithm recursively explores all possible horizontal and vertical cuts, memoizing the results to avoid redundant computations.

1.1 Subproblem Definition

The subproblem involves determining the maximum profit for each possible size of the marble slab. It aims to maximize profit by cutting the slab into smaller rectangles while considering the spot prices of each piece.

1.2 Recurrence of the Subproblem

The recurrence relation for the maximum profit function is as follows:

$$\text{maxProfit}(m,n) = \max \begin{cases} \text{maxProfit}(i,n) + \text{maxProfit}(m-i,n), & \text{for horizontal cuts} \\ \text{maxProfit}(m,j) + \text{maxProfit}(m,n-j), & \text{for vertical cuts} \\ \text{price of the slab itself}, & \text{if no cuts are made} \end{cases}$$

Where i iterates over possible horizontal cuts and j iterates over possible vertical cuts.

1.3 Specific Subproblem(s) Solving the Actual Problem

The algorithm recursively explores all possible combinations of horizontal and vertical cuts to determine the maximum profit. It considers base cases when the slab size reduces to 1×1.

2 Complexity Analysis

2.1 Time Complexity

Time complexity of taking price from the price table is O(1). Time complexity for getting maximum price of one tile is $O(m + n)$ as the loops iterate over possible cuts horizontally going from 1 to m and then vertically 1 to n, resulting in a time complexity of $O(m + n)$. Each tile is computed at most once. The time complexity is determined by the number of subproblems computed, which is proportional to the size of the marble slab (size of the price table) $(m \times n)$. Therefore the resultant time complexity is $O(m^2n + n^2m)$.

2.2 Space Complexity

The space complexity is dominated by the memoization table, which has dimensions $(m + 1) \times (n + 1)$, resulting in $O(m \times n)$ space complexity.

3 Pseudo Code

Algorithm 1 maxProfit function

```
int[][]memo {Memoization table to store max profits with default value 0}
maxProfit P, m, n
if memo[m][n] ≠ 0 then
    return memo[m][n]
else if m == 1 and n == 1 then
    memo[m][n] ← P[0][0]
    return P[0][0]
end if
maxP ← P[m − 1][n − 1]
for i from 1 to m do
    tempProfit ← maxProfit(P, i, n) + maxProfit(P, m − i, n) {total Profit from this cut with recursion call}
    maxP ← max(maxP, tempProfit)
end for
for j from 1 to n do
    tempProfit ← maxProfit(P, m, j) + maxProfit(P, m, n − j) {total Profit from this cut with recursion call}
    maxP ← max(maxP, tempProfit)
end for
memo[m][n] ← maxP
return maxP
```

Algorithm 2 Main Function

```
1: main()
2: // Initialize P and fill the value of P, m, and n
3: P, m, n
4: // Initialize memoization table
5: memo = new int[m + 1][n + 1]
6: maxProfit = maxProfit(P, m, n)
7: print(maxProfit)
```

4 Algorithm

Algorithm 3 maxProfit function

maxProfitP, m, n memo$[m][n] \neq 0$ memo$[m][n]$ $m = 1$ **and** $n = 1$ memo$[m][n] \leftarrow P[0][0]$ $P[0][0]$ maxP $\leftarrow P[m-1][n-1]$ i **from** 1 **to** $m - 1$ tempProfit $\leftarrow$ maxProfit(P, i, n) + maxProfit$(P, m - i, n)$ total Profit from this cut maxP $\leftarrow$ max(maxP, tempProfit) j **from** 1 **to** $n - 1$ tempProfit $\leftarrow$ maxProfit(P, m, j) + maxProfit$(P, m, n - j)$ maxP $\leftarrow$ max(maxP, tempProfit) memo$[m][n] \leftarrow$ maxP maxP