ADA-2024: Homework-3

Deadline: 20th February, 2024; Full Marks: 25

Guidelines for submission: If your algorithm is based on dynamic programming, then you must write your solution using the following format in that order.

- Subproblem definition.
- Recurrence of the subproblem.
- The specific subproblem(s) that solves the actual problem.
- Algorithm description.
- Explanation of the running time of your algorithm.

Problem: You have mined a large slab or marble from a quarry. For simplicity, suppose the marble slab is a rectangle measuring n centimeters in height and m centimeters in width. You want to cut the slab into smaller rectangles of integral pieces (i.e. every small rectangle piece should be a cm by b cm dimension) of various sizes. You have a marble saw that can make either horizontal or vertical cuts across any rectangular slab. At any time, you can query the spot price $P[x, y]$ by an x cm by y cm marble rectangle, for any positive integers x and y.

These prices depend on customer demand, and people who buy marble counter tops are weird, so don't make any assumptions about them; in particular, larger rectangles may have significantly smaller spot prices. Given the array of spot prices and the integers m and n as input, describe an algorithm how to subdivide m cm by n cm marble slab into integral pieces to maximize your profit.

Ans. No pre-processing is applicable for this problem. The problem restricts the cut of the slab into only integral pieces. It is therefore natural that the smallest possible single slab is when m = n = 1 cm. Following we give the subproblem, for the given problem.

- **Subproblem Definition: (3 marks)**
  Let's denote $\textsc{Profit}(x, y)$ as the maximum profit possible for a slab of dimension x cm by y cm. $\textsc{Cost}(x, y)$ is the cost of a single slab of dimension x cm by y cm.
- **Recurrence Relation: (8 marks)**
  The recurrence can be defined as follows:

$$\textsc{Profit}(x, y) = \max \begin{cases} \textsc{Cost}(x, y), \\ \max_{1 \le z < x} \left[\textsc{Profit}(z, y) + \textsc{Profit}(x - z, y)\right], \\ \max_{1 \le z < y} \left[\textsc{Profit}(x, z) + \textsc{Profit}(x, y - z)\right] \end{cases}$$

  Here, the value of z belongs to $\{1, 2, \cdots, x - 1\}$ or $\{1, 2, \cdots, y - 1\}$ as applicable.

- **Base Cases: (1 marks)**

  The base cases is PROFIT(1, 1) = COST(1, 1).

- **Subproblem that solves the actual problem: (2 marks)**

  Subproblem that solves the original problem: PROFIT(m, n).

- **Algorithm Description: (8 marks)**

  This algorithm is designed to solve a dynamic programming problem related to maximizing profit from cutting a marble slab. The slab can be subdivided into two integral pieces in each step, either horizontally or vertically. The subproblem PROFIT(x, y) represents the maximum profit that can be obtained from a slab of size x cm by y cm. The cost of a single slab of a particular dimension is represented by COST(x, y).

  The recurrence relation PROFIT(x, y) is defined as the maximum of three values: the cost of the slab itself, the sum of profits from cutting the slab horizontally into two pieces, and the sum of profits from cutting the slab vertically into two pieces. The value of z used for the cuts ranges from 1 to $x - 1$ or $y - 1$, depending on the direction of the cut.

  The base case is defined as PROFIT(1, 1) = COST(1, 1), which means the maximum profit from a slab of size 1 cm by 1 cm is the cost of the slab itself.

  The subproblem that solves the original problem is PROFIT(m, n), which represents the maximum profit that can be obtained from the original slab of size m cm by n cm. This solution is obtained by solving smaller subproblems and using their solutions to solve larger ones, following the principle of dynamic programming. The algorithm iterates over all possible cuts for each slab size, updating the maximum profit value for each size, until it computes the maximum profit for the original slab size. This approach ensures that all possible subdivisions of the slab are considered, and the maximum profit is found.

  The algorithm can be described as follows:

```
def max_profit(m, n, cost):
    //Initilize the table
    for i in range(1, m):
        for j in range(1, n):
            profit[i][j] = 0

    //Filling up the table
    for i in range(1, m):
        for j in range(1, n):
            max_val = cost[i][j]
            for k in range(1, i-1):
                max_val = max(max_val, profit[k][j] + profit[i-k][j])
            for k in range(1, j-1):
                max_val = max(max_val, profit[i][k] + profit[i][j-k])
            profit[i][j] = max_val

    return profit[m][n]
```

Note: In practice one can define the range of k to be $\{1, \cdots, \lceil \frac{i}{2} \rceil\}$ (or $\{1, \cdots, \lceil \frac{j}{2} \rceil\}$) in the for loop condition. It is correct, because, as an example, observe that $[1][j]$ and

$[i-(i-1)][j]$ indicates the same cell of the table. Thus, when the value of $k = 1$ and $k = (i-1)$, the above pseudocode is accessing the same cell twice, which is unnecessary. However, this does not improve over the running time of the algorithm. So, to keep the algorithm simple, did not implement the efficiency.

- **Running Time: (3 marks)**

  The running time of this algorithm is $O(mn(m+n))$. This is because there are m × n subproblems, and we spend O(m + n) time for each subproblem to find the maximum profit