Group Member:
Name: Guoqing Zhang
Email: zhang.g@ufl.edu
Gihub Account: eddiezgq
GitHub repository link: https://github.com/eddiezgq/Final_Project_Stock_Profit_Maximition

4 Milestone 1: Understanding the problems-solutions

4.1 Problem 1

Output:
(1, 2, 5, 15)
Explanation: Choosing the first stock (1-based indexing). Buying it in day 2nd day and selling it on the fifth day (1-based indexing), yields the maximum profit of 15 (sell price 16 minus buy price 1).

4.2 Problem 2

Output:
[(1, 3, 5), (2, 1, 3), (4, 1, 2)]
Explanation: Performing at most 3 transactions, selling the first stock on the 5th day after buying on 1st day, 2nd stock selling on the 3rd day buying on 1st day, 4th stock selling on 2nd day buying on 1st day
yields the maximum profit of 100 (transaction 1: 50 − 15 = 35, transaction 2: 30 − 10 = 20, transaction 3: 50 − 5 =45)

4.3 Problem 3

Output:
[(3, 1, 3), (2, 6, 7)]
Explanation: To achieve the maximum profit, buy 3rd stock on day 1, sell it on day 3. buy 2nd stock on day 6 and sell it on day 7 adhering to 2 days waiting period.
yields the maximum profit of 11 (transaction 1: 9 − 5 = 4, transaction 2: 8 − 1 = 7)

PROJECT TITLE	Final Project	**COURSE NAME**	COP4533
PROJECT MANAGER	Guoqing Zhang	**DATE**	6/15/25

WBS NUMBER	TASK TITLE	TASK OWNER	START DATE	DUE DATE	DURATION	PCT OF TASK COMPLETE	PHASE ONE															PHASE TWO														
							WEEK 1					WEEK 2					WEEK 3					WEEK 4					WEEK 5					WEEK 6				
							M	T	W	R	F	M	T	W	R	F	M	T	W	R	F	M	T	W	R	F	M	T	W	R	F	M	T	W	R	F
1	**Milestone 1 Understanding the problems**																																			
1.1	Problem 1	Guoqing Zhang	6/8/25	6/15/25	7	100%					🟦	🟦	🟦																							
1.2	Problem 2	Guoqing Zhang	6/8/25	6/15/25	7	100%								🟦	🟦	🟦	🟦	🟦	🟦																	
1.3	Problem 3	Guoqing Zhang	6/8/25	6/15/25	7	100%									🟦	🟦	🟦	🟦	🟦	🟦																
1.4	TODO	Guoqing Zhang	6/8/25	6/15/25	7	100%											🟦	🟦	🟦	🟦																
2	**Milestone 2: Algorithm Design**	Guoqing Zhang																																		
2.1	Task - 1	Guoqing Zhang	6/16/25	7/20/25	4	22%																▨	▨	▨	▨											
2.2	Task - 2	Guoqing Zhang	6/16/25	7/20/25	4	16%																				▨	▨	▨	▨	▨	▨					
2.3	Task - 3	Guoqing Zhang	6/16/25	7/20/25	4	0%																														
2.4	Any two tasks between task - 4, task - 5, task - 6, task - 7.	Guoqing Zhang	6/16/25	7/20/25	4	0%																														

2.5	TODO	Guoqing Zhang	6/16/25	7/20/25	4	0%
3	**Milestone 3: Algorithm Implementation**	Guoqing Zhang	7/21/25	8/3/25	**14**	0%
4	**Presentation**	Guoqing Zhang	7/29/25	8/3/25	**4**	0%

Milestone 2

1 Project Description:

In this final project, you will have the opportunity to design and implement some algorithms to solve variations of Stock buy sell to maximize profit problem. The problem set is divided into different tasks, each with its own unique objectives and requirements.

1.1 Problem Statement - 1:

You are given a matrix A of dimensions $m \times n$, where each element represents the predicted prices of m different stocks for n consecutive days. Your task is to determine a single transaction involving the purchase and sale of a single stock that yields the maximum profit.

1.1.1 Input:

- A matrix A of dimensions $m \times n$ $(1 \leq m, n \leq 1000)$, where each element $A[i][j]$ $(0 \leq A[i][j] \leq 10^5)$ represents the predicted price of the i-th stock on the j-th day. Rows represent stocks, columns represent days.

1.1.2 Output:

Return a tuple $(i, j_1, j_2,$**profit**$)$ where:

- i $(1 \leq i \leq m)$ is the index of the chosen stock.
- j_1 $(1 \leq j_1 \leq n)$ is the day when you would sell the stock to maximize profit.
- j_2 $(1 \leq j_2 \leq n)$ is the day when you would sell the stock to maximize profit.
- profit is the maximum profit achievable through this single transaction (sell price minus buy price).

1.1.3 Constraints:

1. You can only perform one transaction, which involves buying and selling a single stock. You can only sell the stock you bought previously.
2. You must buy before selling; in other words, j_1 must be less than j_2.
3. The prices are non-negative integers and represent the stock's value at a specific time.
4. You want to maximize profit, so you're looking for the highest possible difference between the sell price and the buy price.
5. If no profitable transaction is possible, return a tuple $(0, 0, 0, 0)$ to indicate this.

1.1.4 Example:

Input:

$$A = \begin{bmatrix} 7 & 1 & 5 & 3 & 6 \\ 2 & 4 & 3 & 7 & 9 \\ 5 & 8 & 9 & 1 & 2 \\ 9 & 3 & 14 & 8 & 7 \end{bmatrix}$$

Output:

$$(4, 2, 3, 11)$$

Explanation: Choosing the 4th stock (1-based indexing). Buying it in day 2nd day and selling it on the 3rd day (1-based indexing), yields the maximum profit of 11 (sell price 14 minus buy price 3).

1.2 Problem Statement - 2:

You are given a matrix A of dimensions $m \times n$, where each element represents the predicted prices of m different stocks for n consecutive days. Additionally, you are given an integer k $(1 \leq k \leq n)$. Your task is to find a sequence of at most k transactions, each involving the purchase and sale of a single stock, that yields the maximum profit.

1.2.1 Input:

- A matrix A of dimensions $m \times n$ $(1 \leq m, n \leq 1000)$, where each element $A[i][j]$ $(0 \leq A[i][j] \leq 10^5)$ represents the predicted price of the i-th stock on the j-th day.
- An integer k $(1 \leq k \leq n)$ representing the maximum number of transactions allowed.

1.2.2 Output:

Return a sequence of tuples (i, j_1, j_2) representing the transactions where:

- i $(1 \leq i \leq m)$ is the index of the chosen stock.
- j_1 $(1 \leq j_1 \leq n)$ is the day when you would buy the stock to maximize profit in this transaction.
- j_2 $(1 \leq j_2 \leq n)$ is the day when you would sell the stock to maximize profit in this transaction.

1.2.3 Constraints:

1. You can perform at most k transactions, each involving the purchase and sale of a single stock.
2. You must buy before selling; in other words, j_1 must be less than j_2.
3. The prices are non-negative integers and represent the stock's value at a specific time.
4. You want to maximize profit, so you're looking for the highest possible difference between the sell price and the buy price.
5. If no profitable transaction is possible, return an empty sequence. If there are multiples sequences achieving the same optimal profit, return any one of them.

1.2.4 Example:

Input:

$$A = \begin{bmatrix} 7 & 1 & 5 & 3 & 6 \\ 2 & 9 & 3 & 7 & 9 \\ 5 & 8 & 9 & 1 & 6 \\ 9 & 3 & 4 & 8 & 7 \end{bmatrix}$$

$$k = 3$$

Output:

$$[(2, 1, 2), (1, 2, 3), (2, 3, 5)]$$

Explanation: Performing at most 3 transactions, selling the 2nd stock on the 2nd day after buying on 1st day, 1st stock selling on the 3rd day buying on 2nd day, 2nd stock selling on 5th day buying on 3rd day yields the maximum profit of 17 (transaction 1: $9 - 2 = 7$, transaction 2: $5 - 1 = 4$, transaction 3: $9 - 3 = 6$). **Note**: In each transaction tuple *(i, j_1, j_2)*, the buy day j_1 is strictly less than the sell day j_2. However, when comparing different transactions, we do not consider them overlapping as along as the actual days involved do not conflict. For example, (1,2) and (2,3) are not considered as overlapping because the sell day of the first is the same as the buy day of the second. Similarly, (2,3) and (3,5) are valid together. But if it is (2,4) and (3,5), then we consider them as overlapping.

1.3 Problem Statement - 3:

You are given a matrix A of dimensions $m \times n$, where each element represents the predicted prices of m different stocks for n consecutive days. Additionally, you are given an integer c $(1 \leq c \leq n - 2)$. Your task is to find the maximum profit achievable by trading stocks, subject to the restriction that you cannot buy any stock for c days after selling any stock. If you sell a stock on day j_2, you are not allowed to buy any stock until day $j_2 + c + 1$ in the next transaction.

Input:

- A matrix A of dimensions $m \times n$ $(1 \le m, n \le 1000)$, where each element $A[i][j]$ $(0 \le A[i][j] \le 10^5)$ represents the predicted price of the i-th stock on the j-th day.
- An integer c $(1 \le c \le n-2)$ representing the waiting period after selling a stock before buying another.

Output:

Return a sequence of tuples (i, j_1, j_2) representing the transactions where:

- i $(1 \le i \le m)$ is the index of the chosen stock.
- j_1 $(1 \le j_1 \le n)$ is the day when you would buy the stock to maximize profit in this transaction.
- j_2 $(1 \le j_2 \le n)$ is the day when you would sell the stock to maximize profit in this transaction.

1.3.1 Example:

Input:

$$A = \begin{bmatrix} 2 & 9 & 8 & 4 & 5 & 0 & 7 \\ 6 & 7 & 3 & 9 & 1 & 0 & 8 \\ 1 & 7 & 9 & 6 & 4 & 9 & 11 \\ 7 & 8 & 3 & 1 & 8 & 5 & 2 \\ 1 & 8 & 4 & 0 & 9 & 2 & 1 \end{bmatrix}$$

$$c = 2$$

Output:

$$[(3, 1, 3), (2, 6, 7)]$$

Explanation: To achieve the maximum profit, buy 3rd stock on day 1, sell it on day 3. buy 2nd stock on day 6 and sell it on day 7 adhering to 2 days waiting period.

Task 1: Brute Force Algorithm (O(m × n²))

Variable Definitions:

- `m`, `n`: number of stocks and days
- `A[m][n]`: price matrix
- `max_profit`, `best_stock`, `buy_day`, `sell_day`: track result

Pseudocode

```
function BruteForceMaxProfit(A: matrix of integers, m: int, n: int):

    Initialize max_profit ← 0
    Initialize best_stock ← -1
    Initialize buy_day ← -1
    Initialize sell_day ← -1

    for stock from 0 to m - 1:
        for buy from 0 to n - 2:
            for sell from buy + 1 to n - 1:
                profit ← A[stock][sell] - A[stock][buy]
                if profit > max_profit:
                    max_profit ← profit
                    best_stock ← stock
                    buy_day ← buy
                    sell_day ← sell

    if max_profit > 0:
        return (best_stock, buy_day, sell_day, max_profit)
    else:
        return (0, 0, 0, 0)
```

Task 2: Greedy Algorithm (O(m × n))

Variable Definitions:

- `min_price`: lowest price seen so far for this stock
- `min_day`: day corresponding to `min_price`
- `max_profit`: highest profit seen so far

Pseudocode

```
function GreedyMaxProfit(A: matrix of integers, m: int, n: int):

    Initialize max_profit ← 0
    Initialize best_stock ← -1
    Initialize buy_day ← -1
    Initialize sell_day ← -1

    for stock from 0 to m - 1:
        min_price ← A[stock][0]
        min_day ← 0

        for day from 1 to n - 1:
            current_price ← A[stock][day]
            profit ← current_price - min_price

            if profit > max_profit:
                max_profit ← profit
                best_stock ← stock
                buy_day ← min_day
                sell_day ← day

            if current_price < min_price:
                min_price ← current_price
                min_day ← day

    if max_profit > 0:
        return (best_stock, buy_day, sell_day, max_profit)
    else:
        return (0, 0, 0, 0)
```

Task 3: Dynamic Programming Algorithm (O(m × n))

Variable Definitions:

- `dp[i][j]`: max profit for stock `i` ending at day `j`
- `min_price`: current min price for stock `i`
- Final answer is the max `dp[i][j]` across all stocks and days

Pseudocode

```
function DPMaxProfit(A: matrix of integers, m: int, n: int):

    Initialize max_profit ← 0
    Initialize best_stock ← -1
    Initialize buy_day ← -1
    Initialize sell_day ← -1

    for stock from 0 to m - 1:
        min_price ← A[stock][0]
        min_day ← 0
        dp[0] ← 0  // max profit on day 0 is 0

        for day from 1 to n - 1:
            profit ← A[stock][day] - min_price
            dp[day] ← max(dp[day - 1], profit)

            if profit > max_profit:
                max_profit ← profit
                best_stock ← stock
                buy_day ← min_day
                sell_day ← day

            if A[stock][day] < min_price:
                min_price ← A[stock][day]
                min_day ← day

    if max_profit > 0:
        return (best_stock, buy_day, sell_day, max_profit)
    else:
        return (0, 0, 0, 0)
```

Summary Table

Task	Method	Time Complexity	Description
1	Brute Force	$O(m \times n^2)$	Try all buy-sell pairs

2	Greedy	O(m × n)	Track min price so far
3	DP-inspired	O(m × n)	Like greedy, but using DP style

Task 4 – DP Algorithm for Problem 2

Time Complexity: O(m × n² × k)

Variable Definitions:

- `A[m][n]`: stock prices
- `k`: max allowed transactions
- `dp[t][d]`: max profit using up to `t` transactions up to day `d`
- `profit[i][j1][j2]`: profit from buying stock `i` on day `j1` and selling on day `j2`

Pseudocode:

```
function MaxKTransactionsDP(A: matrix of integers, m: int, n: int, k:
int):
    Initialize profit_list ← empty list

    // Step 1: Precompute all profitable transactions
    for stock from 0 to m - 1:
        for buy from 0 to n - 2:
            for sell from buy + 1 to n - 1:
                profit ← A[stock][sell] - A[stock][buy]
                if profit > 0:
                    profit_list.append((stock, buy, sell, profit))

    Sort profit_list by sell day
    // Step 2: DP table: dp[t][d] = max profit using t transactions up
to day d
    Initialize dp[0][*] ← 0 for all days
    for t from 1 to k:
        for d from 0 to n - 1:
            dp[t][d] ← dp[t][d - 1] // do nothing on day d
            for (stock, buy, sell, prof) in profit_list:
                if sell == d:
                    previous_day ← buy - 1
                    if previous_day ≥ 0:
                        dp[t][d] ← max(dp[t][d], dp[t -
1][previous_day] + prof)
                    else:
                        dp[t][d] ← max(dp[t][d], prof)

    return dp[k][n - 1] // max profit using at most k transactions
```

Task 6 – DP Algorithm for Problem 3 (Waiting Period c)

Variable Definitions:

- `A[m][n]`: price matrix
- `c`: cooldown period after selling
- `dp[d]`: max profit up to day `d`
- `last_sell`: last day a stock was sold

Pseudocode:

```
function MaxProfitWithCooldown(A: matrix of integers, m: int, n: int,
c: int):
    Initialize dp[0...n] ← 0
    Initialize transactions ← empty list
    for buy_day from 0 to n - 2:
        for sell_day from buy_day + 1 to n - 1:
            for stock from 0 to m - 1:
                profit ← A[stock][sell_day] - A[stock][buy_day]
                if profit > 0:
                    // Ensure cooldown
                    if buy_day ≥ 0:
                        cooldown_okay ← True
                        for (s, b, s_day) in transactions:
                            if b ≤ buy_day ≤ s_day + c:
                                cooldown_okay ← False
                        if cooldown_okay:
                            if dp[sell_day] < dp[buy_day] + profit:
                                dp[sell_day] ← dp[buy_day] + profit
                                transactions.append((stock, buy_day,
sell_day))

    return transactions
```

Summary

Task	Problem	Approach	Complexity
4	P2	DP with full scan	$O(m \cdot n^2 \cdot k)$
6	P3	DP with cooldown constraint	$O(m \cdot n^2)$

The programming language to be used in the implementation of the algorithms: Python

import math

def max_profit_stock_trading(A, c):
"""
Calculates the maximum profit achievable from stock trading with a waiting period.

Milestone 3

Problem 1: Maximum Profit from a Single Transaction

Algorithm Design (Optimized)

The problem asks to find a single transaction (a buy and a sell) across all stocks that yields the maximum profit. The core idea is to find the lowest price to buy a stock and the highest price to sell it on a later day. Since a single transaction is allowed across *all* stocks, we need to compare the best possible profit for each individual stock.

A naive brute-force approach would be to check every possible buy day `j1` and every possible sell day `j2 > j1` for every stock `i` and calculate the profit, leading to a time complexity of O(mcdotn2).

A more efficient, optimized algorithm can solve this in linear time with respect to the total number of data points in the matrix, O(mcdotn). For each stock, we can iterate through the days once, keeping track of the minimum price encountered so far. As we iterate through the days, we calculate the potential profit by subtracting the current minimum price from the current day's price. If this profit is the highest we've seen so far across all stocks, we update our maximum profit and store the details of that transaction (stock index, buy day, sell day, and profit).

This optimized approach works because for any given sell day, the maximum profit is always achieved by buying at the lowest possible price on a previous day. By keeping a running minimum, we avoid the nested loop for the buy day, reducing the complexity.

Code:

```
def solve_problem_1(A):
    """
    Finds the maximum profit from a single transaction on any stock.

    Args:
        A (list of list of int): A matrix where A[i][j] is the price
of stock i on day j.
                                 (0-indexed internally, but logic
adapts to 1-indexed problem statement)

    Returns:
        tuple: (stock_idx, buy_day, sell_day, profit) or (0,0,0,0) if
no profit.
    """
    m = len(A)  # Number of stocks
    if m == 0:
        return (0, 0, 0, 0)
    n = len(A[0]) # Number of days
    if n < 2:
        return (0, 0, 0, 0)

    max_profit = 0
    best_stock_idx = 0
    best_buy_day = 0
    best_sell_day = 0

    # Iterate through each stock
    for i in range(m):
        min_price = A[i][0]
        buy_day_for_current_stock = 1  # 1-based index for the day we
would buy

        # Iterate through the days for the current stock
        for j in range(1, n):
            current_price = A[i][j]
            current_profit = current_price - min_price
            # If the current profit is a new maximum, update the
overall best transaction
            if current_profit > max_profit:
                max_profit = current_profit
                best_stock_idx = i + 1  # 1-based index
                best_buy_day = buy_day_for_current_stock
                best_sell_day = j + 1  # 1-based index

            # If the current price is the new minimum, update our
potential buy day
            # for future transactions on this stock
            if current_price < min_price:
                min_price = current_price
                buy_day_for_current_stock = j + 1

    # If no profitable transaction was found, return (0, 0, 0, 0)
    if max_profit <= 0:
        return (0, 0, 0, 0)

    return (best_stock_idx, best_buy_day, best_sell_day, max_profit)
```

Design Rationale:

- **Problem:** Find the single best buy/sell transaction across all stocks and all days to maximize profit. A buy must happen before a sell.
- **Key Insight:** For any given sell day, the maximum profit is achieved by buying the stock at the absolute lowest price on any *preceding* day.
- **Algorithm:** We can iterate through each stock one by one. For each stock, we make a single pass through its daily prices. We maintain a variable min_price to keep track of the lowest price seen so far for that specific stock. On each day, we calculate the potential profit by subtracting the current min_price from the current day's price. If this new profit is greater than our overall max_profit, we update the max_profit and store the details of this transaction (stock, buy day, sell day). If the current day's price is lower than our min_price, we update min_price and the potential buy day.

- **Time Complexity:** The algorithm iterates through m stocks and n days for each stock. This leads to a time complexity of O(m · n), which is highly efficient. A naive brute-force approach would be O(m · n2), which is much slower for large inputs.
- **Code Explanation:**
    - The solve_problem_1 function takes the price matrix A as input.
    - It initializes max_profit to 0 and variables to store the best transaction details.
    - The outer loop for i in range(m) iterates through each stock.
    - The inner loop for j in range(1, n) iterates through the days, starting from the second day to allow for a transaction.
    - min_price and buy_day_for_current_stock track the best buying opportunity for the current stock being considered.
    - The if current_profit > max_profit block updates the overall best result.
    - The if current_price < min_price block updates the best buy day for the current stock, preparing for potential future transactions on that same stock.
    - The function returns a tuple representing the best transaction, using 1-based indexing as specified in the problem. If no profitable transaction is found, it correctly returns (0, 0, 0, 0).

Assumptions:

1. **Positive Prices:** The prices of the stocks are assumed to be non-negative.
2. **Valid Input:** The input is a non-empty 2D array (matrix) where `A[i][j]` represents the price of stock `i` on day `j`. It is assumed that the number of days `n` is at least 2 for a profitable transaction to be possible.
3. **Single Transaction:** Only one buy and one sell transaction is permitted across the entire set of stocks and days.
4. **No Short Selling:** The algorithm assumes you must buy before you can sell, as per standard stock market transactions.

5. **Instantaneous Transaction:** The time it takes to buy and sell is assumed to be instantaneous, and there are no transaction fees or commissions.

Limitations:

1. **No Negative Profits:** The code returns a profit of 0 and no transaction if the best possible scenario involves a loss. This is a design choice that assumes the user would not make a losing transaction.
2. **Ambiguous Buy/Sell Days:** If there are multiple pairs of buy/sell days that yield the same maximum profit, the algorithm will return the last one it encountered. The problem statement does not specify a tie-breaking rule.
3. **Simplified Model:** This solution does not account for real-world complexities such as transaction fees, taxes, or market liquidity.

Problem 2: Maximum Profit from at most k Transactions

Algorithm Design (Dynamic Programming)

This problem is a generalization of the single-transaction problem. We are allowed to make at most `k` transactions, and they cannot overlap. This problem can be solved efficiently using dynamic programming.

We'll define a 2D DP table, `dp[i][j]`, to represent the maximum profit we can achieve using at most `i` transactions up to day `j` (0-indexed). The table size will be (k+1)timesn.

The recurrence relation for `dp[i][j]` is based on two choices:

1. Do not perform a transaction on day `j`: In this case, the maximum profit is the same as the maximum profit up to day `j-1` using `i` transactions, which is `dp[i][j-1]`.
2. Perform a transaction that ends on day `j`: This requires finding the best day `p` to buy a stock (`p < j`). The profit from this transaction would be `A[stock_idx][j] - A[stock_idx][p]`. The profit from all previous transactions would have been calculated up to day `p`, using at most `i-1` transactions. So, we need to find the `max(dp[i-1][p] + A[stock_idx][j] - A[stock_idx][p])` for all possible `p` and `stock_idx`.

The transition can be summarized as: `dp[i][j] = max(dp[i][j-1], max(A[stock_idx][j] - A[stock_idx][p] + dp[i-1][p]))` for `0 <= p < j` and all `stock_idx`.

This can be optimized. We can maintain a running maximum of `(dp[i-1][p] - A[stock_idx][p])` for `p` up to `j-1` to avoid the innermost loop, reducing the complexity from O(kcdotn2cdotm) to O(kcdotncdotm).

To reconstruct the path, we will need to store a "parent" pointer or similar information for each `dp[i][j]` state that indicates which transaction led to that maximum profit. We can then backtrack from `dp[k][n-1]` to build the list of transactions.

Code:

```
def solve_problem_2(A, k):
    """
    Finds the maximum profit from at most k non-overlapping
transactions.

    Args:
        A (list of list of int): A matrix where A[i][j] is the price
of stock i on day j.
        k (int): The maximum number of transactions allowed.
    Returns:
        list of tuple: A sequence of transactions (stock_idx, buy_day,
sell_day)
                       to maximize total profit.
    """
    m = len(A)
    if m == 0 or k == 0:
        return []
    n = len(A[0])
    if n < 2:
        return []

    # DP state: dp[i][j] = max profit using at most i transactions up
to day j
    dp = [[0] * n for _ in range(k + 1)]

    # parent[i][j] stores the transaction (stock_idx, buy_day,
sell_day) that
    # resulted in the max profit at dp[i][j].
    parent = [[None] * n for _ in range(k + 1)]

    # Iterate through each transaction count
    for i in range(1, k + 1):
        # max_diff helps in optimizing the inner loop.
        # max_diff = max(dp[i-1][p] - A[stock_idx][p]) for all p < j
        max_diff = [float('-inf')] * m

        # Iterate through each day
        for j in range(n):
            # Option 1: Don't perform a transaction on day j.
            # Max profit is same as the previous day.
            if j > 0:
                dp[i][j] = dp[i][j - 1]
                parent[i][j] = parent[i][j - 1]

            # Option 2: Perform a transaction ending on day j.
            # Find the best buy day for each stock for this
transaction.
            for stock_idx in range(m):
                current_profit = A[stock_idx][j] + max_diff[stock_idx]

                # Check if this new transaction gives a better profit
for this day
                if current_profit > dp[i][j]:
                    dp[i][j] = current_profit
                    # To reconstruct the path, we need to find the buy
day.
                    # We can iterate backward from j-1 to find the day
p that created max_diff
                    best_buy_day = -1
                    for p in range(j):
                        if dp[i-1][p] - A[stock_idx][p] ==
max_diff[stock_idx]:
                            best_buy_day = p
                            break
                    if best_buy_day != -1:
                        parent[i][j] = (stock_idx + 1, best_buy_day +
1, j + 1)

            # Update max_diff for the next day's calculations
            for stock_idx in range(m):
                max_diff[stock_idx] = max(max_diff[stock_idx],
dp[i-1][j] - A[stock_idx][j])

    # Reconstruct the transaction path by backtracking from the max
profit cell
    transactions = []
    curr_k = k
    curr_day = n - 1

    # Find the optimal k and day to start the reconstruction from
    max_profit_overall = 0
    optimal_k = 0
    optimal_day = 0
    for i in range(k + 1):
        for j in range(n):
            if dp[i][j] > max_profit_overall:
                max_profit_overall = dp[i][j]
                optimal_k = i
                optimal_day = j

    curr_k = optimal_k
    curr_day = optimal_day

    while curr_k > 0 and curr_day >= 0:
        if parent[curr_k][curr_day] is not None and \
           (curr_day == 0 or parent[curr_k][curr_day] !=
parent[curr_k][curr_day-1]):
            # A new transaction was made on this day. Add it to the
list.
            stock_idx, buy_day, sell_day = parent[curr_k][curr_day]
            transactions.insert(0, (stock_idx, buy_day, sell_day))

            # Jump back to the day before this transaction started.
            curr_day = buy_day - 2
            curr_k -= 1
        else:
            # No new transaction, move to the previous day
            curr_day -= 1

    return transactions
```

Design Rationale:

- Problem: Find the maximum profit from at most `k` non-overlapping transactions across all stocks and days.
- Key Insight: This is a classic dynamic programming problem. The optimal solution for `i` transactions up to day `j` can be derived from the optimal solutions for fewer transactions on previous days.
- Algorithm:

    - We use a 2D DP table, `dp[i][j]`, to store the maximum profit with at most `i` transactions up to day `j`.
    - The recurrence relation considers two options for each state `(i, j)`:
        1. Don't make a transaction on day `j`: The profit is the same as the previous day, `dp[i][j-1]`.
        2. Sell a stock on day `j` as part of the `i`-th transaction: To find the best buy day for this transaction, we must consider all previous days `p < j`. The profit from this transaction would be `price[j] - price[p]`, which is added to the maximum profit we had on day `p` with `i-1` transactions, `dp[i-1][p]`. We want to maximize `dp[i-1][p] - price[p]` and then add `price[j]`. We can pre-calculate and store `max(dp[i-1][p] - price[p])` for all `p` up to `j-1` to make this step efficient.
- **Time Complexity:** The nested loops for `k` transactions, `n` days, and `m` stocks give a time complexity of $O(k \cdot n \cdot m)$. This is a significant improvement over a brute-force approach.
- **Code Explanation:**
    - The `solve_problem_2` function takes the price matrix `A` and the maximum number of transactions `k` as input.
    - It initializes a DP table `dp` and a `parent` table to store the transaction that led to the maximum profit. This is crucial for reconstructing the final list of transactions.
    - The outer loop `for i in range(1, k + 1)` iterates through the number of allowed transactions.
    - The inner loop `for j in range(n)` iterates through the days.
    - `dp[i][j] = dp[i][j-1]` handles the case where no transaction is performed on day `j`.

- The `max_diff` array is a key optimization. It keeps track of the maximum value of `dp[i-1][p] - A[stock_idx][p]` for all days `p < j`. This allows us to calculate the best profit for a transaction ending on day `j` in O(m) time instead of O(m·n).
- The code then checks if a new transaction ending on day `j` yields a better profit and updates `dp[i][j]` and the `parent` table accordingly.
- After filling the DP table, a separate `while` loop is used to backtrack through the `parent` table from the cell with the highest total profit (`dp[optimal_k][optimal_day]`) to reconstruct the sequence of transactions. This ensures we return the actual transactions, not just the total profit.

Assumptions:

1. **Valid Input:** The input is a non-empty 2D array (matrix) of stock prices and a non-negative integer `k` representing the maximum number of transactions.
2. **Positive Edge Weights (Prices):** The DP approach assumes that stock prices are non-negative.
3. **Non-overlapping Transactions:** A new transaction can only begin after a previous one has been completed. The code correctly handles this by using the `dp[i-1][p]` state, which represents the profit *before* the current transaction.
4. **Instantaneous Transaction:** Similar to Problem 1, transactions are assumed to be instantaneous and free of fees.

Limitations:

1. **Computational Complexity:** The time complexity of O(k·n·m) can be a limitation for very large `k`, `n`, or `m`. For a very large number of days or stocks, this could be slow.
2. **No Tie-Breaking Rule:** If multiple sequences of transactions result in the same maximum profit, the algorithm will return the first one it

finds. The specific sequence returned may depend on the implementation details and iteration order.
3. **Memory Usage:** The DP table `dp` and the `parent` table have a size of (k+1)×n, which can consume a significant amount of memory for large values of `k` and `n`.
4. **Simplified Model:** The model does not consider transaction costs, which would require a modification to the DP recurrence to subtract fees for each transaction.

Problem 3
Code:

```
import math

def max_profit_stock_trading(A, c):
    """
    Calculates the maximum profit achievable from stock trading with a
waiting period.

    Args:
        A (list of list of int): A matrix where A[i][j] is the price
of stock i on day j.
                                 (0-indexed internally, but logic
adapts to 1-indexed problem statement)
        c (int): The waiting period after selling a stock before
buying another.

    Returns:
        list of tuple: A sequence of transactions (stock_idx, buy_day,
sell_day)
                       to maximize total profit. Days and stock
indices are 1-based
                       as per the problem statement.
    """
    m = len(A)  # Number of stocks
    n = len(A[0]) # Number of days

    # dp[j] will store the maximum profit achievable up to day j
(0-indexed).
    # Initialize with 0, as no profit can be made initially.
    dp = [0] * n

    # parent[j] will store the transaction that leads to the maximum
profit at day j.
    # This is used for reconstructing the optimal path.
    # Each entry will be (stock_idx, buy_day_idx, sell_day_idx) or
None.
    parent = [None] * n

    # Iterate through each possible selling day (j2 in problem
statement, corresponds to day_idx in 0-indexed)
    for day_idx in range(n):
        # Initialize dp[day_idx] based on the previous day's maximum
profit.
        # This means we either don't make a transaction ending on
day_idx,
        # or we find a better profit by doing so.
        if day_idx > 0:
            dp[day_idx] = dp[day_idx - 1]
            parent[day_idx] = parent[day_idx - 1]

        # Iterate through each stock
        for stock_idx in range(m):
            # Iterate through each possible buying day (j1 in problem
statement, corresponds to prev_day_idx)
            for prev_day_idx in range(day_idx):
                # Ensure we can actually make a profit
                if A[stock_idx][day_idx] > A[stock_idx][prev_day_idx]:
                    current_transaction_profit = A[stock_idx][day_idx]
- A[stock_idx][prev_day_idx]

                    # Calculate the index for the previous profit
considering the waiting period
```

```
                    # If we buy on prev_day_idx, the previous
transaction must end at or before:
                    # prev_day_idx - c - 1
                    # (j1 in problem statement is 1-indexed,
prev_day_idx is 0-indexed)
                    # (j2 in problem statement is 1-indexed, day_idx
is 0-indexed)
                    # (sell_day + c + 1 for next BUY)
                    # So, if we buy on prev_day_idx (0-indexed), the
last sell day before this buy
                    # must be <= (prev_day_idx - 1)
                    # and that last sell day + c + 1 <= prev_day_idx
                    # so, prev_sell_day <= prev_day_idx - c - 1

                    # The maximum profit from previous transactions
must be from a day
                    # that allows buying on prev_day_idx given the 'c'
day cooldown.
                    # If we sell on 'prev_prev_day_idx', then next buy
can be from 'prev_prev_day_idx + c + 1'
                    # So, if current buy day is 'prev_day_idx', then
previous sell day 'prev_prev_day_idx' must be
                    # 'prev_prev_day_idx + c + 1 <= prev_day_idx' =>
'prev_prev_day_idx <= prev_day_idx - c - 1'

                    prev_profit_consider_day_idx = prev_day_idx - c -
1

                    prev_total_profit = 0
                    if prev_profit_consider_day_idx >= 0:
                        prev_total_profit =
dp[prev_profit_consider_day_idx]

                    total_profit_for_this_path =
current_transaction_profit + prev_total_profit

                    # If this path yields a greater profit for day_idx
                    if total_profit_for_this_path > dp[day_idx]:
                        dp[day_idx] = total_profit_for_this_path
                        # Store the transaction that led to this max
profit at day_idx
                        # Store 0-indexed stock and days
                        parent[day_idx] = (stock_idx, prev_day_idx,
day_idx)

    # Reconstruct the sequence of transactions
    transactions = []
    current_day_idx = n - 1 # Start from the last day

    while current_day_idx >= 0 and parent[current_day_idx] is not
None:
        stock_idx, buy_day_idx, sell_day_idx = parent[current_day_idx]

        # Check if this transaction was actually the one that
*defined* dp[sell_day_idx]
        # and not just an inherited value from an earlier day.
        # This check is crucial for accurate path reconstruction when
dp[day_idx] inherits from dp[day_idx-1]

        # Calculate the profit if this transaction was made
        current_transaction_profit = A[stock_idx][sell_day_idx] -
A[stock_idx][buy_day_idx]

        prev_profit_consider_day_idx = buy_day_idx - c - 1
        prev_total_profit_expected = 0
        if prev_profit_consider_day_idx >= 0:
            prev_total_profit_expected =
dp[prev_profit_consider_day_idx]

        if dp[sell_day_idx] == (current_transaction_profit +
prev_total_profit_expected):
            # If the current dp value was indeed set by this
transaction
            # Add to the beginning of the list to maintain
chronological order
            # Convert to 1-based indexing for output
            transactions.insert(0, (stock_idx + 1, buy_day_idx + 1,
sell_day_idx + 1))
            current_day_idx = prev_profit_consider_day_idx # Move to
the day before the required cooldown
        else:
            # This means dp[current_day_idx] was inherited from
dp[current_day_idx - 1]
            # without a new transaction ending on current_day_idx
being optimal.
            # So, we just move back one day to look for the actual
transaction that defined it.
            current_day_idx -= 1
            # Need to re-evaluate parent[current_day_idx] if it was
just inherited.
            # This is why simple `parent[current_day_idx] =
parent[current_day_idx - 1]`
            # in the DP loop might be tricky for reconstruction.
            # A more robust reconstruction might iterate back from
`n-1` to 0, and if `parent[i]` is not None
            # and `parent[i]` is the transaction that ACTUALLY
achieved `dp[i]`, then add it and jump back.

    # Revised reconstruction logic for clarity and correctness:
    # Iterate backwards from the last day, finding the optimal
transactions.
    final_transactions = []
    current_profit_target = dp[n - 1]

    # Find the last day where a transaction occurred.
    actual_last_transaction_day = n - 1
    while actual_last_transaction_day >= 0 and
parent[actual_last_transaction_day] is None:
        actual_last_transaction_day -= 1

    if actual_last_transaction_day < 0: # No transactions found
        return []

    curr_idx = actual_last_transaction_day
    while curr_idx >= 0:
        if parent[curr_idx] is not None:
            stock_idx, buy_day_idx, sell_day_idx = parent[curr_idx]

            # Check if this transaction *actually* resulted in
dp[sell_day_idx]
            # and wasn't just inherited from dp[sell_day_idx - 1]
            transaction_profit = A[stock_idx][sell_day_idx] -
A[stock_idx][buy_day_idx]

            prev_profit_idx_for_check = buy_day_idx - c - 1
            prev_dp_value = 0
            if prev_profit_idx_for_check >= 0:
                prev_dp_value = dp[prev_profit_idx_for_check]

            # If this transaction is what led to the current
dp[curr_idx] (or dp[sell_day_idx])
            # This logic needs to be careful: parent[curr_idx] stores
the best transaction *ending* on curr_idx.
            # It might be `dp[curr_idx] = max(dp[curr_idx-1],
profit_from_new_transaction + prev_dp)`
            # So, we need to check if the current `parent[curr_idx]`
was the one that led to the value `dp[curr_idx]`
            # or if `dp[curr_idx]` was simply inherited from
`dp[curr_idx-1]`.

            # The simplified reconstruction below is usually
sufficient for this type of DP.
            # Start from the end, if parent[i] exists, it means a
transaction ended at 'i'.
            # Add it, then jump back to `buy_day - c - 1` to find the
next one.

            # Let's use a standard path reconstruction for DP.
            # Walk back from the end. If parent[i] is defined, it
means *a* transaction ended here.
```

```
            # Then jump back to the day before the cooldown of that
transaction.

            if dp[curr_idx] > (dp[curr_idx - 1] if curr_idx > 0 else
0): # Check if dp[curr_idx] was updated by a transaction ending at
curr_idx
                 final_transactions.insert(0, (stock_idx + 1,
buy_day_idx + 1, sell_day_idx + 1))
                 curr_idx = prev_profit_idx_for_check # Jump back to
the day before the cooldown allows a new purchase
            else:
                 # If dp[curr_idx] was not updated by a transaction
ending on curr_idx, it means
                 # dp[curr_idx] == dp[curr_idx - 1]. So, we move to
the previous day.
                 curr_idx -= 1
        else:
            # If parent[curr_idx] is None, it means no transaction
ended on this day.
            # Move to the previous day.
            curr_idx -= 1

    return final_transactions

# --- Test Cases ---

# Example from Problem 3.1:
# Input:
# A = [[2, 9, 8, 4, 5, 0, 7],
#      [6, 7, 3, 9, 1, 0, 8],
#      [1, 7, 9, 6, 4, 9, 11], # Corrected: My previous manual trace
used [5,8,9,1,2,3,10] which was a typo from a different similar
problem.
#      [7, 8, 3, 1, 8, 5, 2],
#      [1, 8, 4, 0, 9, 2, 1]]
# c = 2
# Expected Output: [(3, 1, 3), (2, 6, 7)]
# Let's use the A matrix provided in the image for 1.3 Example
A_example = [
    [2, 9, 8, 4, 5, 0, 7],  # Stock 1
    [6, 7, 3, 9, 1, 0, 8],  # Stock 2
    [1, 7, 9, 6, 4, 9, 11], # Stock 3 (this is the one with 1,7,9...
which matches the example output)
    [7, 8, 3, 1, 8, 5, 2],  # Stock 4
    [1, 8, 4, 0, 9, 2, 1]   # Stock 5
]
c_example = 2

# Manually verify the example's output again with the provided
A_example:
# Transaction 1: (Stock 3, Day 1, Day 3)
# Stock 3 (index 2): [1, 7, 9, 6, 4, 9, 11]
# Buy Day 1 (index 0): A[2][0] = 1
# Sell Day 3 (index 2): A[2][2] = 9
# Profit = 9 - 1 = 8.
# (Note: My previous manual trace used A[2][0]=5, A[2][2]=9, profit=4,
which was incorrect as per THIS input matrix).

# Transaction 2: (Stock 2, Day 6, Day 7)
# Stock 2 (index 1): [6, 7, 3, 9, 1, 0, 8]
# Buy Day 6 (index 5): A[1][5] = 0
# Sell Day 7 (index 6): A[1][6] = 8
# Profit = 8 - 0 = 8.

# Total profit for example output: 8 + 8 = 16.

# Let's run the code with the example
print("--- Running Example Test Case ---")
result_example = max_profit_stock_trading(A_example, c_example)
print(f"Input Matrix A:\n{A_example}")
print(f"Waiting Period c: {c_example}")
print(f"Calculated Output: {result_example}")
print(f"Expected Output: [(3, 1, 3), (2, 6, 7)]")

# --- Additional Test Cases ---
# Test Case 1: Simple case, no waiting period (c=0)
A_test1 = [[10, 1, 100]]
c_test1 = 0
# Expected: [(1, 2, 3)] or [(1,1,3)] if we can buy on day 1 (price 1)
and sell on day 3 (price 100)
# A[0][0]=10, A[0][1]=1, A[0][2]=100
# Buy on Day 2 (index 1), Sell on Day 3 (index 2) -> Profit = 100 - 1
= 99
print("\n--- Running Test Case 1 (c=0) ---")
result_test1 = max_profit_stock_trading(A_test1, c_test1)
print(f"Input Matrix A:\n{A_test1}")
print(f"Waiting Period c: {c_test1}")
print(f"Calculated Output: {result_test1}")
print(f"Expected Output: [(1, 2, 3)]")

# Test Case 2: Multiple transactions possible, with c
A_test2 = [
    [10, 20, 5, 30, 10, 40], # Stock 1
    [1, 2, 3, 4, 5, 6]       # Stock 2
]
c_test2 = 1 # Sell on j1, cannot buy until j1 + 1 + 1 = j1 + 2

# Possible transactions for Stock 1:
# (S1, D1, D2): 20-10=10. Cooldown until D2+1+1 = D4.
#   Then can buy on D4. Max Profit = 10 + (40-10)=30. Path: (1,1,2),
(1,5,6)
# (S1, D3, D4): 30-5=25. Cooldown until D4+1+1 = D6.
#   Then can buy on D6.
# (S1, D3, D6): 40-5=35. This is one transaction.
# For Stock 2:
# (S2, D1, D6): 6-1=5

print("\n--- Running Test Case 2 ---")
result_test2 = max_profit_stock_trading(A_test2, c_test2)
print(f"Input Matrix A:\n{A_test2}")
print(f"Waiting Period c: {c_test2}")
print(f"Calculated Output: {result_test2}")
```

```
# Expected output might be complex, let's trace:
# dp values:
# Day 0: dp[0]=0
# Day 1: dp[1]=0
# Day 2: (S1,D1,D2) profit=10. dp[2]=10, parent[2]=(0,0,1)
# Day 3: (S2,D1,D2) profit=1. dp[3]=10.
# Day 4: (S1,D3,D4) profit=25. prev_profit_idx = 3-1-1=1. dp[1]=0.
Total=25. dp[4]=max(10,25)=25. parent[4]=(0,2,3)
# Day 5: dp[5]=25
# Day 6: (S1,D3,D6) profit=35. prev_profit_idx=2-1-1=1. dp[1]=0.
Total=35.
#       (S1,D5,D6) profit=30. prev_profit_idx=4-1-1=2. dp[2]=10.
Total=30+10=40. This is higher!
#       So for Day 6 (index 5): (S1,D5,D6) is
A[0][5]-A[0][4]=40-10=30.
#       prev_buy_idx=4. cooldown to prev_buy_idx-c-1 = 4-1-1 = 2.
#       dp[2] is 10. So total profit = 30 + 10 = 40.
#       dp[5]=max(25, 40)=40. parent[5]=(0,4,5) -> (Stock 1, Buy Day
5, Sell Day 6)
#       Path: (S1,D5,D6). Previous day to consider: index 2. parent[2]
is (0,0,1).
# Final Path: [(1, 1, 2), (1, 5, 6)] Total Profit: (20-10) + (40-10) =
10 + 30 = 40.
print(f"Expected Output: [(1, 1, 2), (1, 5, 6)]") # This trace matches
```

Explanation of the Code and Design Rationale:

1. Dynamic Programming Approach:
   - The core of the solution is a dynamic programming array `dp` where `dp[j]` stores the maximum profit achievable by considering transactions that *end on or before* day `j`.
   - `parent[j]` is an auxiliary array used to reconstruct the sequence of transactions. `parent[j]` stores the details of the specific transaction (stock index, buy day, sell day) that resulted in `dp[j]`.
2. State Definition (`dp[day_idx]`):
   - For each `day_idx` from `0` to `n-1`, `dp[day_idx]` is computed.

   - `dp[day_idx]` can be either:
      - The same as `dp[day_idx - 1]` (if no new optimal transaction ends on `day_idx`).
      - Or, it's the profit from a new transaction ending on `day_idx`, plus the maximum profit accumulated from previous transactions that respect the waiting period `c`.
3. Iteration and Calculation:
   - The outer loop iterates through `day_idx` (potential selling days).
   - Inside, it iterates through all `stock_idx` and then all `prev_day_idx` (potential buying days before `day_idx`).
   - For each potential transaction `(stock_idx, prev_day_idx, day_idx)`:
      - Calculate `current_transaction_profit = A[stock_idx][day_idx] - A[stock_idx][prev_day_idx]`.
      - Determine `prev_profit_consider_day_idx`: This is the last day a previous transaction could have ended to allow buying on `prev_day_idx` after the `c` day cooldown. If you sell on `X`, you can next buy on `X + c + 1`. So if you buy on `prev_day_idx`, the previous sell must have been at `X <= prev_day_idx - c - 1`.
      - `prev_total_profit` is retrieved from `dp[prev_profit_consider_day_idx]`. If `prev_profit_consider_day_idx` is negative, it means no prior transactions were possible or needed, so `prev_total_profit` is 0.
      - `total_profit_for_this_path = current_transaction_profit + prev_total_profit`.

            ■ If `total_profit_for_this_path` is greater than `dp[day_idx]`, update `dp[day_idx]` and `parent[day_idx]`.

4. Path Reconstruction:
    - The `parent` array is crucial for reconstruction. It's a common pattern in DP.
    - We start from the last day (`n - 1`) and work backward.
    - If `dp[curr_idx]` is greater than `dp[curr_idx - 1]` (or `0` if `curr_idx` is `0`), it implies that a transaction ending on `curr_idx` *contributed* to this maximum profit.
    - We add the transaction stored in `parent[curr_idx]` to our `final_transactions` list (inserting at the beginning to maintain chronological order).
    - Then, we "jump back" in time to `buy_day_idx - c - 1` to find the previous transaction, effectively skipping the cooldown period.
    - If `dp[curr_idx]` is not greater than `dp[curr_idx - 1]`, it means the profit for this day was simply inherited from the previous day without a new transaction. So, we just move `curr_idx` back by `1`.
5. Indexing:
    - The problem statement uses 1-based indexing for stock indices and days.
    - The Python code uses 0-based indexing internally for arrays (e.g., `A[stock_idx][day_idx]`).
    - The final output converts indices back to 1-based to match the problem's required format.

Assumptions and Limitations:

- Assumes `A` is a non-empty matrix with consistent row lengths.
- Assumes `c` is within the valid range as specified (`1 <= c <= n-2`).

- Edge cases like `n=1` or `n=2` might need special handling if `c` range was different, but `c <= n-2` ensures `n` is at least 3.
- The algorithm correctly handles cases where no profitable transactions can be made (returns an empty list).

Citation:

Kleinberg, J., & Tardos, E. (2006). *Algorithm design*. Pearson/Addison Wesley.