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 $(i, j_1, j_2, \text{profit})$ 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_i, you are not allowed to buy any stock until day $j_i + c + 1$ in the next transaction.

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 c $(1 \leq c \leq n-2)$ 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 $(1 \leq i \leq m)$ 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.

1.3.1 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