COP4533 – Final Project

Final Project Report Structure (Milestone 3)

STOCK TRADING ALGORITHMS

using

Brute Force, Greedy,

And Dynamic Programming Techniques

Submitted by:

Loubna BENCHAKOUK

l.benchakouk@ufl.edu

GitHub Username: LoubnaB023

Repository Link: https://github.com/loubnaB023/COP4533--FinalProject

Date of submission: 08/02/2025

UNIVERSITY OF FLORIDA

Table of Contents

Milestone 1: Understanding the problems

GitHub Repository Link: https://github.com/loubnaB023/COP4533--FinalProject

Team members:

- *Name:* Loubna Benchakouk
  Email: l.benchakouk@ufl.edu
  GitHub Username: loubnaB023

- *Name:* Anhelina Liashynska
  Email: aliashynska@ufl.edu
  GitHub Username: Angellsh

- *Name:* Jacob Ramos
  Email: jacob.ramos@ufl.edu
  GitHub Username: JacobR678

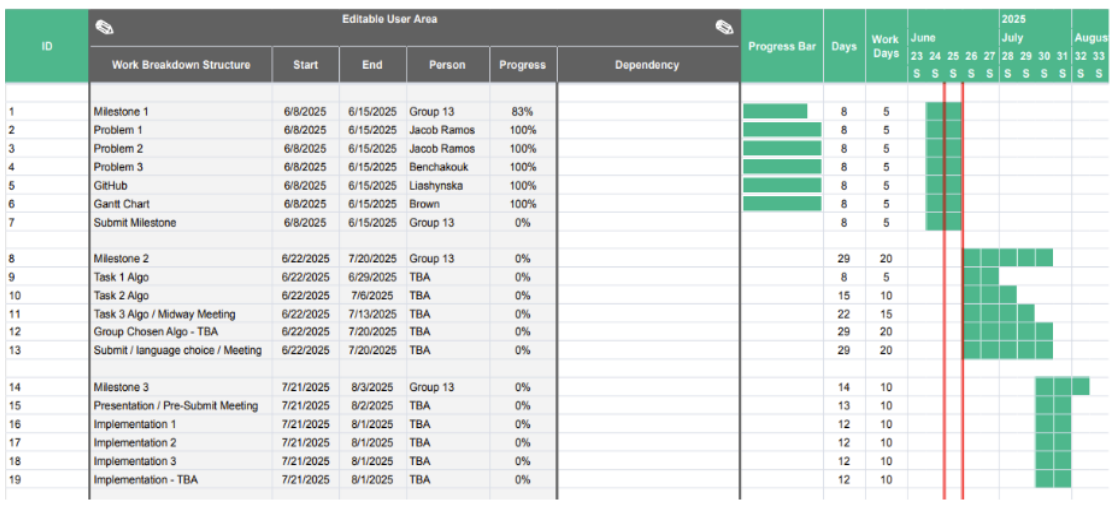
- *Name:* Dani Brown
  Email: brown.d@ufl.edu
  GitHub Username: danBrownGithub

Member Roles:

- **Jacob Ramos**: Problem 1 & 2 completed both tasks (100% progress)
- **Loubna Benchakouk**: Problem 3 completed (100% progress)
- **Anhelina Liashynska**: GitHub setup
- **Dani Brown**: Gantt Chart
- Roles for upcoming milestones will be assigned later

Communication methods: We use Discord for regular communication and Google Docs for document collaboration.

Project Gantt Chart:



ID	Work Breakdown Structure	Start	End	Person	Progress	Dependency	Progress Bar	Days	Work Days
	Editable User Area								
1	Milestone 1	6/8/2025	6/15/2025	Group 13	83%			8	5
2	Problem 1	6/8/2025	6/15/2025	Jacob Ramos	100%			8	5
3	Problem 2	6/8/2025	6/15/2025	Jacob Ramos	100%			8	5
4	Problem 3	6/8/2025	6/15/2025	Benchakouk	100%			8	5
5	GitHub	6/8/2025	6/15/2025	Liashynska	100%			8	5
6	Gantt Chart	6/8/2025	6/15/2025	Brown	100%			8	5
7	Submit Milestone	6/8/2025	6/15/2025	Group 13	0%			8	5
8	Milestone 2	6/22/2025	7/20/2025	Group 13	0%			29	20
9	Task 1 Algo	6/22/2025	6/29/2025	TBA	0%			8	5
10	Task 2 Algo	6/22/2025	7/6/2025	TBA	0%			15	10
11	Task 3 Algo / Midway Meeting	6/22/2025	7/13/2025	TBA	0%			22	15
12	Group Chosen Algo - TBA	6/22/2025	7/20/2025	TBA	0%			29	20
13	Submit / language choice / Meeting	6/22/2025	7/20/2025	TBA	0%			29	20
14	Milestone 3	7/21/2025	8/3/2025	Group 13	0%			14	10
15	Presentation / Pre-Submit Meeting	7/21/2025	8/2/2025	TBA	0%			13	10
16	Implementation 1	7/21/2025	8/1/2025	TBA	0%			12	10
17	Implementation 2	7/21/2025	8/1/2025	TBA	0%			12	10
18	Implementation 3	7/21/2025	8/1/2025	TBA	0%			12	10
19	Implementation - TBA	7/21/2025	8/1/2025	TBA	0%			12	10

1.1 Problem 1

We are given a matrix of stock prices where each row represents a different stock and each column will represent a different day. We calculate the maximum potential profit for each specific stock using a 1-based index.

Each stock/day combination maximum profit: (1,2,5,15) (2,1,3,9) (3,1,2,2) (4,2,5,7)

Answer: Stock/Day Combination Maximum Profit: (1,2,5,15)

Explanation:

- Stock 1 yields the maximum when bought on the 2nd day and sold on the 5th day for a profit of 15.
- Stock 2 yields the maximum profit when bought on day 1 and sold on day 3 yielding a profit of 9.
- Stock 3 yields the maximum potential profit when bought on day 1 and sold on day 2 yielding a profit of 2.
- Finally, stock 4 yields the maximum potential profit when bought on day 2 and sold on day 5 yielding a profit of 7.

1.2 Problem 2

We are given a matrix where each row represents a different stock and each column will represent a different day. We are given an integer k which will represent the maximum number of non-overlapping transactions permitted, in this case k = 3. For each transaction we must buy and sell one stock.

Answer: (4,1,2), (2,2,3), (1,3,5) total profit = 90

Explanation:

1. Stock 4: Buy on the 1st day at price 5, sell on the 2nd day at price 50 for a profit of 45.
2. Stock 2: Buy on the 2nd day at price 20, sell on the 3rd day at price 30 for a profit of 10.
3. Stock 1: Buy on the 3rd day at price 15, sell on the 5th day at price 50 for a profit of 35.
4. Total profit = 45 + 10 + 35 = 90

1.3 Problem 3

Problem Statement

We are given a matrix where each row represents a different stock and each column will represent a different day. Additionally, we are given an integer c which will represent a cooldown period where we cannot buy any stock for c days after selling any stock. If a stock is sold on day i, the next stock will not be eligible for purchase until day i + c + 1. For this example, c = 2.

Answer: (3,1,3), (3,6,7) total profit = 4 + 7 = 11

Explanation:

1. First transaction we buy stock 3 on day 1 and sell on day 3 for a profit of 4
2. Since the stock was sold on day 3 we cannot purchase another stock till day 6
3. On day 6, we buy stock 3 again and sell on day 7 for a profit of 7
4. The total profit is 11

Transaction rules:

1. We can only buy before we sell, and only once per transaction.
2. Resting period: after we sell on day j2 we need to wait until (j2+c+1) day to buy.
3. We can perform multiple transactions on any stock while following the cooldown rule.
4. Main objective is to maximize the total profit across all valid transactions.

Input:

We have a matrix A where each:
Row = one stock
Column = one day
A[i][j] = price of stock(i + 1) on day(j + 1)
Matrix A:

Day	1	2	3	4	5	6	7
Stock_1	7	1	5	3	6	8	9
Stock_2	2	4	3	7	9	1	8
Stock_3	5	8	9	1	2	3	10
Stock_4	9	3	4	8	7	4	1
Stock_5	3	1	5	8	9	6	4

Cooldown period: c = 2

To solve this problem we need to find all profitable transactions for each stock(row in the matrix)

1. Choose a buy day and then try all sell days that come after that buy day
2. For each(buy, sell) day, check if the price on the sell day is higher than the price on the buy day.
3. Keep just the profitable pairs(i, j, l)

Step 1: Identify All Possible Profitable Transactions

For each stock, we need to check all (buy, sell) pairs where buyDay < sellDay and profit > 0:

Stock 1: [7, 1, 5, 3, 6, 8, 9]

BuyDay	SellDay	BuyPrice	SellPrice	Profit	NextBuy	ValidTransaction
1	2	7	1	-6		
	3	7	5	-2		
	4	7	3	-4		
	5	7	6	-1		
	6	7	8	1	Day9(6+2+1)	No
	7	7	9	2	Day10(7+2+1)	No
2	3	1	5	4	Day6(3+2+1)	(6,7)
	4	1	3	2	Day7(4+2+1)	(7,7)
	5	1	6	5	Day8(5+2+1)	No
	6	1	8	7	Day9(6+2+1)	No
	7	1	9	8	Day10(7+2+1)	No
3	4	5	3	-2		
	5	5	6	1	Day8(5+2+1)	No
	6	5	8	3	Day9	No
	7	5	9	4	Day10	No
4	5	3	6	3	Day8	No
	6	3	8	5	Day9	No
	7	3	9	6	Day10	No
5	6	6	8	2	Day9	No
	7	6	9	3	Day10	No
6	7	8	9	1	Day10	No

From the table we see that the best combination for Stock 1: (2,7) with profit = 8

Stock 2: [2, 4, 3, 7, 9, 1, 8]

BuyDay	SellDay	BuyPrice	SellPrice	Profit	NextBuy	ValidTransaction
1	2	2	4	2	Day5(2+2+1)	(5, 6); (5, 7); (6, 7)
	3		3	1	Day6	(6, 7)
	4		7	5	Day7	(7, 7)
	5		9	7	Day8	No
	6		1	-1		
	7		8	6	Day10	No
2	3	4	3	-1		
	4		7	3	Day7	(7, 7)
	5		9	5	Day8	No
	6		1	-3		
	7		8	4	Day10	No
3	4	3	7	4	Day7	(7, 7)
	5		9	6	Day8	No
	6		1	-2		
	7		8	5	Day10	No
4	5	7	9	2	Day8	No
	6		1	-6		
	7		8	1	Day10	No
5	6	9	1	-8		
	7		8	-1		
6	7	1	8	7	Day9	No

Best single transaction for Stock 2: (1,5) with profit = 7

Stock 3: [5, 8, 9, 1, 2, 3, 10]

BuyDay	SellDay	BuyPrice	SellPrice	Profit	NextBuy	ValidTransaction
1	2	5	8	3	Day5	(5, 6); (5, 7); (6, 7)
	3		9	4	Day6	(6, 7)
	4		1	-4		
	5		2	-3		
	6		3	-2		
	7		10	5	Day10	No
2	3	8	9	1	Day6	(6, 7)
	4		1	-7		
	5		2	-6		
	6		3	-5		
	7		10	2	Day10	No
3	4	9	1	-8		
	5		2	-7		
	6		3	-6		
	7		10	1	Day10	No
4	5	1	2	1	Day8	No
	6		3	2	Day9	No
	7		10	9	Day10	No
5	6	2	3	1	Day9	No
	7		10	8	Day10	No
6	7	3	10	7	Day10	No

Best single transaction for Stock 3: (4,7) with profit = 9

Stock 4: [9, 3, 4, 8, 7, 4, 1]

BuyDay	SellDay	BuyPrice	SellPrice	Profit	NextBuy	ValidTransaction
1	2	9	3	-6		
	3		4	-5		
	4		8	-1		
	5		7	-2		
	6		4	-5		
	7		1	-8		
2	3	3	4	1	Day6(3+2+1)	(6, 7)
	4		8	5	Day7	(7, 7)
	5		7	4	Day8	No
	6		4	1	Day9	No
	7		1	-2		
3	4	4	8	4	Day7	(7, 7)
	5		7	3	Day8	No
	6		4	0	Day9	No
	7		1	-3		
4	5	8	7	-1		
	6		4	-4		
	7		1	-7		
5	6	7	4	-3		
	7		1	-6		
6	7	4	1	-3		

Best single transaction for Stock 4: (2,4) with profit = 5

Stock 5: [3, 1, 5, 8, 9, 6, 4]

BuyDay	SellDay	BuyPrice	SellPrice	Profit	NextBuy	ValidTransaction
1	2	3	1	-2		
	3		5	2	Day6(3+2+1)	(6, 7)
	4		8	5	Day7	(7, 7)
	5		9	6	Day8	No
	6		6	3	Day9	No
	7		4	1	Day10	No
2	3	1	5	4	Day6	(6, 7)
	4		8	7	Day7	(7, 7)
	5		9	8	Day8	No
	6		6	5	Day9	No
	7		4	3	Day10	No
3	4	5	8	3	Day7	(7, 7)
	5		9	4	Day8	No
	6		6	1	Day9	No
	7		4	-1		
4	5	8	9	1	Day8	No
	6		6	-2		
	7		4	-4		
5	6	9	6	-3		
	7		4	-5		
6	7	6	4	-2		

Best single transaction for Stock 5: (2,5) with profit = 8

Since we know the best individual transactions per stock. Now we check if we can combine some of them to build a valid sequence.

Starting with stock 1, the best transaction is: buy on day 2, sell on day 7 with profit = 8. After applying the cooldown rule the next valid buy day is day 10 but our max day is 7. Therefore, we can't combine it with any other transaction

- ⇨ Sequence (1, 2, 7) with total profit = 8

Stock 2: The best transaction is to buy on day 1 and sell on day 5 with profit = 7 and since the next buy day is day 8 we can't make an extra transaction.

- ⇨ Sequence (2, 1, 5) with total profit = 7

but we have another transaction with a smaller profit of 2 if we buy on day 1 and sell on day 2, after the resting period we can buy stock 3 on day 5, sell on day 7 with profit =8

- ⇨ Sequence (2, 1, 2), (3, 5, 7) with total profit = 10

Stock 3 we found that the best transaction is to buy on day 4, sell on day 7 with profit = 9 and since we need to wait for day10 (invalid) to make another transaction

- ⇨ Sequence (3, 4, 7) with total profit = 9

But if we buy on day 1 and sell on day 3 with profit = 4, we can combine it with Stock 2 on day 6 after the cooldown period, we buy on day 6 and sell on day 7 with profit = 7

- ⇨ Sequence (3, 1, 3), (2, 6, 7) with total profit = 11

Stock 4, we have the best profit = 5 if we buy on day 2 and sell on day 4, since the next valid buy day is day 7 and there is no available transaction starting day 7

- ⇨ Sequence (4, 2, 4) with total profit = 5

For Stock 5 the best transaction is when we buy on day 2 and sell on day 5 with profit = 8, after applying the cooldown rule, we don't get a valid day

- ⇨ Sequence (5, 2, 5) with total profit = 8

From the above, the maximum profit = 11 from the sequence (3, 1, 4), (2, 6, 7)

- ⇨ 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

Milestone 2: Algorithm Design

GitHub Repository Link: https://github.com/loubnaB023/COP4533--FinalProject

Individual submission:

- *Name:* Loubna Benchakouk
  Email: l.benchakouk@ufl.edu
  GitHub Username: loubnaB023

2.1 Task-1: Brute Force Algorithm for Problem1 $O(m \cdot n^2)$.

The goal is to find the maximum profit from a single buy/sell transaction on the same stock with one buy before one sell, only one transaction, and return stock index, buy day, sell day, and max profit.

Assumptions & variable definitions:

- m: number of stocks (rows in matrix A)
- n: number of days (columns in A)
- A transaction is defined as buying a stock on day j_1 and selling it later on day j_2, where $j_1 < j_2$.
- The transaction must be on the same stock (row).
- The result should be a tuple: (i, j_1, j_2, profit) where:
  - i: index of the chosen stock (1-based index)
  - j_1: day to buy
  - j_2: day to sell
  - profit = A[i][j_2] - A[i][j_1]

2.1.1 Pseudocode:

```
# --- Pseudocode for MaxProfitBruteForce (Problem-1) Algorithm ---
'''
Algorithm MaxProfitBruteForce(A, m, n)

Input: matrix A(m × n) representing stock prices
Output: tuple (stock, buyDay, sellDay, profit) representing the best stock and days
to buy/sell and max profit

Begin
    // --- initialize variables to store the best result found ---
    maxProfit ← 0
    bestStock ← 0
    bestBuyDay ← 0
    bestSellDay ← 0

    // --- try every possible stock ---
    for i ← 0 to m – 1 do
        // --- try every possible buy day ---
        for j₁ ← 0 to n – 2 do
            // --- try every possible sell day after the buy day ---
            for j₂ ← j₁ + 1 to n – 1 do
                // --- calculate profit for the current transaction ---
                profit ← A[i][j₂] – A[i][j₁]

                // --- if this transaction gives higher profit, update the result -
--
                if profit > maxProfit then
                    maxProfit ← profit
            // 1-based indexing
                    bestStock ← i + 1
                    bestBuyDay ← j₁ + 1
                    bestSellDay ← j₂ + 1
                end if
            end for
        end for
    end for

    // --- return result depending on whether any profit was made ---
    if maxProfit = 0 then
        return (0, 0, 0, 0)   // no profitable transaction found
    else
        return (bestStock, bestBuyDay, bestSellDay, maxProfit)
    end if
End
'''
```

The algorithm checks all possible pairs of days (j_1, j_2) for each stock to calculate potential profit, it loops over every stock (m stocks) and for each stock, compares every pair of buy/ sell days.
The algorithm keeps track of the max profit found so far and stores its stock index and days. If no profitable transaction exists (all negative or 0), it returns (0, 0, 0, 0).

2.2 Task-2: Greedy Algorithm for Problem1 O(m·n)

Assumptions & variable definitions:

- Only one transaction is allowed per stock
- Buy must occur before sell ($j_1 < j_2$)
- Each stock is evaluated independently
- Return (0, 0, 0, 0) if no profit is possible

2.2.1Pseudocode:

```
#--- Pseudocode for MaxProfitGreedySolution (Problem-1) Algorithm ---
'''
Algorithm MaxProfitGreedySolution(A, m, n)

Input: matrix A(m × n) representing stock prices
Output: tuple (stock, buyDay, sellDay, profit) representing the best stock and days
to buy/sell and max profit

Begin
    // ---initialize variables to store best result ---
    maxProfit ← 0
    bestStock ← 0
    bestBuyDay ← 0
    bestSellDay ← 0

    // --- iterate through each stock ---
    for i ← 0 to m – 1 do
        // --- track the minimum price seen so far for this stock ---
        minPrice ← A[i][0]
        minDay ← 0

        // --- scan forward to find best day to sell ---
        for j ← 1 to n – 1 do
            // --- if selling today gives better profit, update result ---
            if A[i][j] – minPrice > maxProfit then
                maxProfit ← A[i][j] – minPrice
                bestStock ← i + 1
                bestBuyDay ← minDay + 1
                bestSellDay ← j + 1
            end if

            // --- If today’s price is lower, update minPrice and minDay ---
            if A[i][j] < minPrice then
                minPrice ← A[i][j]
                minDay ← j
            end if
        end for
    end for

    // --- return result depending on whether profit was made ---
    if maxProfit = 0 then
        return (0, 0, 0, 0)     // no profitable transaction found
    else
        return (bestStock, bestBuyDay, bestSellDay, maxProfit)
    end if
End
'''
```

In this greedy version of the algorithm, we optimize the search for the max profit from a single buy/ sell transaction. For each stock, the algorithm keeps the minimum price observed so far (minPrice) and the corresponding day (minDay). As it iterates through the remaining days, it computes the current potential profit by subtracting minPrice from the price on the current day. If this profit exceeds the previously recorded maxProfit, the algorithm updates the optimal transaction details (stock index, buy day, and sell day). If the current day's price is less than minPrice, it becomes the new minPrice. If no profitable transaction exists, the algorithm returns the default tuple (0, 0, 0, 0).

2.3 Task-3: Dynamic Programming Algorithm for Problem1 O(m·n)

Assumptions & variable definitions:

- m: number of stocks (rows in matrix A)
- n: number of days (columns in A)
- A transaction is defined as buying a stock on day j_1 and selling it later on day j_2, where $j_1 < j_2$.
- The transaction must be on the same stock (row).

- The result should be a tuple: (i, j_1, j_2, profit) where:
  - i: index of the chosen stock (1-based index)
  - j_1: day to buy
  - j_2: day to sell
  - profit = A[i][j_2] - A[i][j_1]

2.3.1 Pseudocode:

```
#--- Pseudocode for MaxProfitDynamicProgramming (Problem-1) Algorithm ---
'''
Algorithm MaxProfitDynamicProgramming(A, m, n)

Input: matrix A(m × n) representing stock prices

Output: tuple (stock, buyDay, sellDay, profit) representing the best stock and days t
o buy/sell and max profit

Begin
    // --- initialize variables ---
    maxProfit ← 0
    bestStock ← 0
    bestBuyDay ← 0
    bestSellDay ← 0

    // --- loop through each stock ---
    for i ← 0 to m – 1 do
        minPrice ← A[i][0]     // minimum price seen so far for stock i
        minDay ← 0             // day when it occurred

        // --- loop through each day for this stock ---
        for j ← 1 to n – 1 do
            currentProfit ← A[i][j] – minPrice

            // --- check if this is the best profit so far ---
            if currentProfit > maxProfit then
                maxProfit ← currentProfit
                bestStock ← i + 1
                bestBuyDay ← minDay + 1
                bestSellDay ← j + 1
            end if

            // --- update minPrice if current day is cheaper ---
            if A[i][j] < minPrice then
                minPrice ← A[i][j]
                minDay ← j
            end if
        end for
    end for

    // --- result ---
    if maxProfit = 0 then
        return (0, 0, 0, 0)       // no profitable transaction found
    else
        return (bestStock, bestBuyDay, bestSellDay, maxProfit)
    end if
End
'''
```

This algorithm uses dynamic programming logic to find the maximum profit from a single transaction per stock. For each stock, it keeps track of the minimum price seen so far (minPrice) and the corresponding day (minDay). As it scans through the days, it computes the profit of selling on the current day minus minPrice and updates the maximum profit and corresponding buy/sell days if it finds a better option. If no profit is possible, the algorithm returns (0, 0, 0, 0).

2.5 Task-5: Dynamic Programming Algorithm for Problem2 O(m·n·k)

Assumptions & variable definitions:

- m: number of stocks (rows in matrix A)
- n: number of days (columns in A)
- k: maximum number of transactions allowed
- A transaction consists of buying a stock on day j_1 and selling it on day j_2, where $j_1 < j_2$
- Transactions can involve different stocks
- Non-overlapping constraint: if the transaction ends on day d, next transaction can start on day d or later
- The result should be a sequence of tuples: [(i_1, j_1, j_2), (i_2, j_3, j_4), ...] representing optimal transactions

2.5.1 Pseudocode:

```
#--- Pseudocode for MultiTransactionStockTrading DP (Problem-2) Algorithm ---
'''
Algorithm: MultiTransactionStockTrading(A, m, n, k)
Input:
A[m × n]: matrix of stock prices (m stocks, n days)
k: maximum number of transactions allowed
Output:
list of transactions (stock, buyDay, sellDay) that maximize profit with at most k transa
ctions

Begin

    // --- DP table to track max profit with t transactions up to day d ---
    dp[0..k][1..n] ← 0       // dp[t][d]: max profit with t transactions by day d

    // --- arrays to track our choices for reconstruction ---
    boughtStock[0..k][1..n]
    boughtDay[0..k][1..n]
    soldStock[0..k][1..n]
    didSell[0..k][1..n] ← false

    // --- initialize result container ---
    transactions ← empty list

    // --- fill DP table ---
    for t ← 1 to k do
        bestBuyProfit ← -A[1][1]            // best profit after buying stock 1 on day 1
        bestBuyStock ← 1
        bestBuyDay ← 1

        for day ← 2 to n do
            // --- case 1: no transaction today, carry forward profit ---
            noSellProfit ← dp[t][day - 1]

            // --- case 2: sell today ---
            maxSellProfit ← -∞
            bestSellStock ← -1

            for stock ← 1 to m do
                profit ← A[stock][day] + bestBuyProfit
                if profit > maxSellProfit then
                    maxSellProfit ← profit
                    bestSellStock ← stock
                end if
            end for

            // --- choose the better of the two options ---
            if maxSellProfit > noSellProfit then
                dp[t][day] ← maxSellProfit
                didSell[t][day] ← true
                soldStock[t][day] ← bestSellStock
                boughtStock[t][day] ← bestBuyStock
                boughtDay[t][day] ← bestBuyDay
            else
                dp[t][day] ← noSellProfit
                didSell[t][day] ← false
            end if

            // --- update best buy opportunity for future sells ---
            for stock ← 1 to m do
                newBuyProfit ← dp[t - 1][day] - A[stock][day]
                if newBuyProfit > bestBuyProfit then
                    bestBuyProfit ← newBuyProfit
                    bestBuyStock ← stock
                    bestBuyDay ← day
                end if
            end for
        end for
    end for

    // --- backtrack to reconstruct the optimal transactions ---
    currentDay ← n
    currentTrans ← k

    while currentDay > 0 and currentTrans > 0 do
        if didSell[currentTrans][currentDay] = true then
            stock ← soldStock[currentTrans][currentDay]
            buyDay ← boughtDay[currentTrans][currentDay]
            transactions.prepend((stock, buyDay, currentDay))
            currentDay ← buyDay - 1
            currentTrans ← currentTrans - 1
        else
            currentDay ← currentDay - 1
        end if
    end while

    return transactions

End
```

This algorithm helps find the best way to make up to k profitable stock trades using prices from m stocks over n days. It builds a table dp[t][d] that keeps track of the highest possible profit at each day, for each number of transactions. At every step, it checks whether it's better to sell today or wait. It remembers when and which stock was bought and sold, so it can later figure out the best trades. In the end, it works backward through the table to list the best trades without any overlap.

2.6 Task-6: Dynamic Programming Algorithm for Problem3 $O(m \cdot n^2)$

The goal here is to maximize total profit from multiple stock transactions with a cooldown period c. After selling a stock on day j_2, the next allowed buy can only happen on day $j_2 + c + 1$ or later.

Assumptions & variable definitions:

Input Variables:

- A[1..m][1..n]: Matrix where A[i][j] represents price of stock i on day j
- m: Number of stocks
- n: Number of days
- c: Cooldown period

State Variables:

- Free[1..n]: Maximum profit on day i when not holding any stock (free to buy)
- Holding[1..n]: Maximum profit on day i when holding a stock (can sell)
- Cooldown[1..n]: Maximum profit on day i when in cooldown (just sold, cannot buy)

Tracking Variables:

- heldStock[1..n]: Which stock we're holding on each day (0 if not holding)
- purchaseDay[1..n]: When we bought the stock we're currently holding
- transactions[]: Final sequence of transactions (stock, buyDay, sellDay)

2.6.1 Pseudocode:

```
#--- Pseudocode for StockTradingWithCooldown DP (Problem-3) Algorithm ---
'''
Algorithm StockTradingWithCooldown(A, m, n, c)

Input: matrix A(m × n) representing stock prices.
        cooldown period c
Output: list of transactions (stock, buyDay, sellDay) maximizing profit with cooldown
constraint

Begin

    // --- DP arrays to track max profit for each state on each day ---
    Free ← -∞          // array [1..n]: not holding any stock, can buy
    Holding ← -∞       // array [1..n]: holding a stock, can sell
    Cooldown ← -∞      // array [1..n]: just sold, in cooldown period

    // --- containers to track our decisions ---
    heldStock ← 0           // array [1..n]: which stock we're holding each day
    purchaseDay ← 0         // array [1..n]: when we bought it

    // --- initialize result container ---
    transactions ← empty list

    // ---- Base case: Day 1, start with no money, no stocks, not in cooldown ---
    Free[1] ← 0              // we start free with 0 profit
    Holding[1] ← -∞          // can't be in hold without buying first
    Cooldown[1] ← -∞         // can't be in cooldown without selling first

    // --- fill the profit arrays day by day ---
    for day ← 2 to n do

        // --- state 1: we are free to buy today ---
        // stay free, do nothing, carry forward yesterday's profit
        Free[day] ← Free[day - 1]

        // cooldown period ended, we can be free again
        if day > c + 1 then           // cooldown lasts c days, so we are free after c
+ 1 days
            if Cooldown[day - 1] > Free[day] then
                Free[day] ← Cooldown[day - 1]
            end if
        end if

        // --- state 2: we are holding a stock today ---
        // we choose to keep holding the same stock from yesterday
        Holding[day] ← Holding[day - 1]
        heldStock[day] ← heldStock[day - 1]           // same stock
        purchaseDay[day] ← purchaseDay[day - 1]       // same purchase date

        // or we buy a new stock today only if we were free yesterday
        for stock ← 1 to m do
            profit ← Free[day - 1] - A[stock][day]    // subtract cost

            if profit > Holding[day] then
                Holding[day] ← profit
                heldStock[day] ← stock           // remember which stock we bought
                purchaseDay[day] ← day           // remember when we bought it
            end if
        end for
```

```
    // --- state 3: we are in cooldown today ---
    // continue cooldown from yesterday
    Cooldown[day] ← Cooldown[day - 1]

    // sell our stock today and enter cooldown
    if heldStock[day - 1] > 0 then     // we were holding something yesterday
        stockToSell ← heldStock[day - 1]
        buyDay ← purchaseDay[day - 1]

        profit ← Holding[day - 1] + A[stockToSell][day]     // add sale price

        if profit > Cooldown[day] then
            Cooldown[day] ← profit
            // record the transaction during reconstruction phase
            transactions.add((stockToSell, buyDay, day))
        end if
    end if

end for

// --- identify the best final state and maximum profit ---
finalMaxProfit ← max(Free[n], Holding[n], Cooldown[n])

// which state gave us the best result?
finalState ← 0      // assuming we ended free

if Holding[n] ≥ Free[n] and Holding[n] ≥ Cooldown[n] then
    finalState ← 1      // we ended holding a stock
else if Cooldown[n] ≥ Free[n] then
    finalState ← 2     // we ended in cooldown
end if

// --- backtrack to find the actual transactions that led to optimal profit ---
transactions.clear()
currentDay ← n
currentState ← finalState

// --- walk backwards through our decisions to reconstruct the optimal path ---
while currentDay > 1 do

    if currentState = 0 then    // we are currently free
        // How did we get to the free state? Two possibilities:
        if currentDay > c + 1 and Free[currentDay] = Cooldown[currentDay - 1] then
            // we came from cooldown
            currentDay ← currentDay - 1
            currentState ← 2
        else
            // we stayed free from the previous day
            currentDay ← currentDay - 1
            currentState ← 0
        end if

    else if currentState = 1 then   // we are currently holding
        // did we buy today or were we already holding?
        boughtToday ← false

        for stock ← 1 to m do
            if Holding[currentDay] = Free[currentDay - 1] - A[stock][currentDay] t
hen
                boughtToday ← true    // found the stock we bought today
                break
            end if
        end for

        if boughtToday then
            // we bought today, so we were free yesterday
            currentDay ← currentDay - 1
            currentState ← 0
        else
            // we were already holding from yesterday
            currentDay ← currentDay - 1
            currentState ← 1
        end if

    else    // currentState = 2, we are in cooldown
        // did we sell today or were we already in cooldown?
        soldToday ← false

        if currentDay > 1 and heldStock[currentDay - 1] > 0 then
            stockSold ← heldStock[currentDay - 1]
            buyDay ← purchaseDay[currentDay - 1]

            // check if selling this stock today gave us our current profit
            if Cooldown[currentDay] = Holding[currentDay - 1] + A[stockSold][curre
ntDay] then
                // if yes, we sold this stock today and record the transaction
                transactions.prepend((stockSold, buyDay, currentDay))
                soldToday ← true
                // we jump back to the day before we bought this stock
                currentDay ← buyDay - 1
                currentState ← 0
            end if
        end if

        if not soldToday then
            // we were already in cooldown from yesterday
            currentDay ← currentDay - 1
            currentState ← 2
        end if

    end if

end while

return transactions

End
```

This algorithm helps to maximize the total profit by allowing multiple stock transactions while respecting a cooldown period c between trades. After selling a stock, we are not allowed to buy another until c days have passed. To manage this, the algorithm uses three dynamic programming arrays: Free, Holding, and Cooldown. Each array tracks the best profit we can achieve on a given day under a specific state. Free[day] represents the maximum profit if we are not holding any stock and are free to buy, Holding[day] keeps track of profits when we are currently holding a stock, and Cooldown[day] represents the profit when we are in a mandatory rest period after selling. Each day, the algorithm evaluates whether to maintain the current state like continuing to hold a stock, or transition like buying or selling a stock. It computes the profit for each action and updates the respective state arrays accordingly. It also remembers which stock was bought or sold and on which day. After going through all the days, it performs a backtracking step, where it walks backward through the Free, Holding, and Cooldown arrays to reconstruct the exact series of buy and sell actions that led to the optimal profit.

Milestone 3: Algorithm Implementation

Section 1: Introduction

In Milestone 3, the goal is to translate the algorithmic pseudocode created in Milestone 2 into actual, working code using an appropriate programming language. The implementation must be tested with appropriate test cases, analyzed for time and space complexity, and documented with any trade-offs or challenges encountered.

1.1 Programming Language Used

The algorithms were implemented using Python, chosen for its simplicity, and readability.

1.2 Tasks Implemented from Milestone 2

In this Milestone, we implemented the following tasks from the ones designed earlier in Milestone 2:

- Task 1: A brute force algorithm for Problem 1 that checks every possible combination of buying and selling to find the biggest profit.
  Time complexity: $O(m \cdot n^2)$
- Task 2: A faster greedy algorithm for Problem 1 that goes through each stock just once to figure out the best day to buy and sell.
  Time complexity: $O(m \cdot n)$
- Task 3: A dynamic programming (DP) version of Problem 1 that improves performance by storing useful results instead of recalculating them.
  Time complexity: $O(m \cdot n)$
- Task 5: A DP algorithm for Problem 2 that finds the best set of stock trades when there's a limit on how many transactions we're allowed to make.
  Time complexity: $O(m \cdot n \cdot k)$
- Task 6: A DP solution for Problem 3 that handles the case where we're required to wait a few days (a cooldown period) after each sell before we can buy again.
  Time complexity: $O(m \cdot n^2)$

Each task in this milestone focuses on maximizing profit under different stock trading scenarios, each with its own set of constraints.

Section 2: Task Implementations

2.1 Task-1: Brute Force Algorithm for Problem1 $O(m \cdot n^2)$.

2.1.1 Problem Recap:

The goal of this task is to find the best single buy and sell transaction for one stock that gives the highest profit. We are given a matrix where each row is a stock, and each column is the price on a specific day. We can only make one transaction, buy on one day and sell on a later day, using the same stock. If no profit is possible, we return (0, 0, 0, 0). The output includes the stock index, buy day, sell day, and the profit. This is solved using a brute-force approach with a time complexity of $O(m \cdot n^2)$

2.1.2 Pseudocode

No changes, Same as submitted in Milestone 2

2.1.3 Python Code

```
def MaxProfitBruteForce(A, m, n):
    """
    Brute force algorithm to find the maximum profit from a single buy/sell trans.

    Parameters:
        A (List[List[int]]): Matrix representing stock prices (m stocks * n days)
        m (int): Number of stocks (rows)
        n (int): Number of days (columns)

    Returns:
        Tuple[int, int, int, int]: (bestStock, bestBuyDay, bestSellDay, maxProfit)
        All values are 1-based indices.
        Returns (0, 0, 0, 0) if no profit is possible.
    """

    # --- edge case: check if the input is empty or missing ---
    if m <= 0 or n <= 1 or not A or n < 2:
        return (0, 0, 0, 0)

    # --- edge case: check if matrix has proper dimensions
    if len(A) != m or any(len(row) != n for row in A):
        return (0, 0, 0, 0)

    # --- initialize variables to store the best result found ---
    maxProfit = 0
    bestStock = 0
    bestBuyDay = 0
    bestSellDay = 0

    # --- try every possible stock ---
    for i in range(m):  # stock index (0-based)
        # --- try every possible buy day ---
        for j1 in range(n - 1):  # buy day
            # --- try every possible sell day after the buy day ---
            for j2 in range(j1 + 1, n):  # sell day
                # --- calculate profit for the current transaction ---
                profit = A[i][j2] - A[i][j1]


# --- if this transaction gives higher profit, update the result ---
                if profit > maxProfit:
                    maxProfit = profit
                    # 1-based index
                    bestStock = i + 1
                    bestBuyDay = j1 + 1
                    bestSellDay = j2 + 1

    # --- return result depending on whether any profit was made ---
    if maxProfit == 0:
        return (0, 0, 0, 0)  # if no profitable transaction found
    else:
        return (bestStock, bestBuyDay, bestSellDay, maxProfit)
```

2.1.4 Test Cases & Output

```python
# --- TEST CASE SECTION ---
if __name__ == "__main__":

    print("=" * 70)
    print("{:^70}".format("BRUTE FORCE ALGORITHM PROBLEM 1 - TASK 1 TESTS"))
    print("=" * 70)

    print("\n Sample test from Project PDF description")
    A = [
        [7, 1, 5, 3, 6],
        [2, 4, 3, 7, 9],
        [5, 8, 9, 1, 2],
        [9, 3, 14, 8, 7]
    ]
    result = MaxProfitBruteForce(A, 4, 5)
    print(f"Expected: (4, 2, 3, 11) | Result: {result}")

    print("\n 1: Mixed stocks with profits")
    matrix1 = [
        [7, 1, 5, 3, 6, 4],
        [2, 8, 3, 9, 1, 5],
        [10, 2, 6, 4, 8, 3]
    ]
    result = MaxProfitBruteForce(matrix1, 3, 6)
    print(f"Expected: (2, 1, 4, 7) | Result: {result}")

    print("\n 2: No profit possible")
    matrix2 = [
        [10, 8, 6, 4, 2],
        [15, 12, 10, 5, 1]
    ]
    result = MaxProfitBruteForce(matrix2, 2, 5)
    print(f"Expected: (0, 0, 0, 0) | Result: {result}")

    print("\n 3: Minimum valid input (1 stock, 2 days)")
    matrix3 = [
        [1, 5]
    ]
    result = MaxProfitBruteForce(matrix3, 1, 2)
    print(f"Expected: (1, 1, 2, 4) | Result: {result}")

    print("\n" + "=" * 70)
    print("{:^70}".format("EDGE CASES"))
    print("=" * 70)

    print("\n 1: Empty matrix")
    result = MaxProfitBruteForce([], 0, 0)
    print(f"Expected: (0, 0, 0, 0) | Result: {result}")

    print("\n 2: One stock One day (no transaction possible)")
    result = MaxProfitBruteForce([[5]], 1, 1)
    print(f"Expected: (0, 0, 0, 0) | Result: {result}")


    print("\n 3: Invalid Matrix dimensions")
    matrix_invalid = [
        [1, 2, 3],
        [4, 5]
    ]
    result = MaxProfitBruteForce(matrix_invalid, 2, 3)
    print(f"Expected: (0, 0, 0, 0) | Result: {result}")

    print("\n 4: All stocks with the same pric  )
    matrix_flat = [
        [3, 3, 3, 3],
        [3, 3, 3, 3]
    ]
    result = MaxProfitBruteForce(matrix_flat, 2, 4)
    print(f"Expected: (0, 0, 0, 0) | Result: {result}")
```

```
File  Edit  Selection  View  Go  Run  ···        Search

PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS  SQL HISTORY  TASK MONITOR        Python Debug Console

PS C:\Users\lbenc\Documents\CS_Degree\Summer_2025\COP_4533 (Algorithms Abstraction and Design)\Milestones> py Task-1_MaxProfitBruteForce.py
======================================================================
            BRUTE FORCE ALGORITHM PROBLEM 1 - TASK 1 TESTS
======================================================================

 Sample test from Project PDF description
Expected: (4, 2, 3, 11) | Result: (4, 2, 3, 11)

 1: Mixed stocks with profits
Expected: (2, 1, 4, 7) | Result: (2, 1, 4, 7)

 2: No profit possible
Expected: (0, 0, 0, 0) | Result: (0, 0, 0, 0)

 3: Minimum valid input (1 stock, 2 days)
Expected: (1, 1, 2, 4) | Result: (1, 1, 2, 4)

======================================================================
                              EDGE CASES
======================================================================

 1: Empty matrix
Expected: (0, 0, 0, 0) | Result: (0, 0, 0, 0)

 2: One stock One day (no transaction possible)
Expected: (0, 0, 0, 0) | Result: (0, 0, 0, 0)

 3: Invalid Matrix dimensions
Expected: (0, 0, 0, 0) | Result: (0, 0, 0, 0)

 4: All stocks with the same price
Expected: (0, 0, 0, 0) | Result: (0, 0, 0, 0)
PS C:\Users\lbenc\Documents\CS_Degree\Summer_2025\COP_4533 (Algorithms Abstraction and Design)\Milestones>
```

2.1.5 Analysis

The brute force algorithm is the most basic way to solve this problem. It works by checking every possible way to buy and sell stocks to find the one that makes the most money. This approach uses three loops nested inside each other. The first loop looks at each stock, the second loop tries each day as a possible buy day, and the third loop tries each day after the buy day as a possible sell day.

Algorithm Process

Here's how the algorithm works step by step. For each stock in our matrix, we consider every possible day to buy that stock. Then, for each buy day, we look at every day that comes after it as a potential sell day. We calculate the profit by subtracting the buy price from the sell price. If this profit is bigger than any profit we've found so far, we save this transaction as our new best option. By the end, we've checked every possible transaction and found the one with the highest profit.

Time and Space Complexity

The time complexity of this algorithm is $O(m \times n^2)$, which means it gets slow very quickly as our data gets bigger. This happens because we have three loops: one for m stocks and two for the days (which gives us n^2). So if we have 1000 stocks and 1000 days, we'd need to do about one billion calculations. The space complexity is O(1) because we only need a few variables to keep track of the best solution we've found so far.

Advantages:

The main advantage of the brute force approach is that it's easy to understand and implement. Anyone can look at the code and immediately see what it's doing. It also guarantees that we'll find the correct answer because we check every possibility.

Limitation:

The big problem with this approach is that it's very slow for large datasets. When we tested it with bigger inputs, it took much longer to run compared to the other algorithms.

Practical Use:

The brute force algorithm works well for small problems or when we're learning about algorithms. It's also useful for checking if our other, more complex algorithms are working correctly. But for real-world applications where we might have thousands of stocks and many days of data, this approach would be too slow to be practical.

2.2 Task-2: Greedy Algorithm for Problem1 O(m·n)

2.2.1 Problem Recap:

This task also focuses on finding the best single buy and sell transaction for one stock, but instead of checking every possible pair, we use a more efficient greedy approach. We are given a matrix where each row is a stock and each column is its price on a specific day. The goal is to go through each stock once and keep track of the lowest price seen so far to find the highest possible profit. Only one transaction is allowed, and buying must happen before selling. If no profit is possible, the result is (0, 0, 0, 0). This approach improves performance with a time complexity of O(m·n).

2.2.2 Pseudocode:

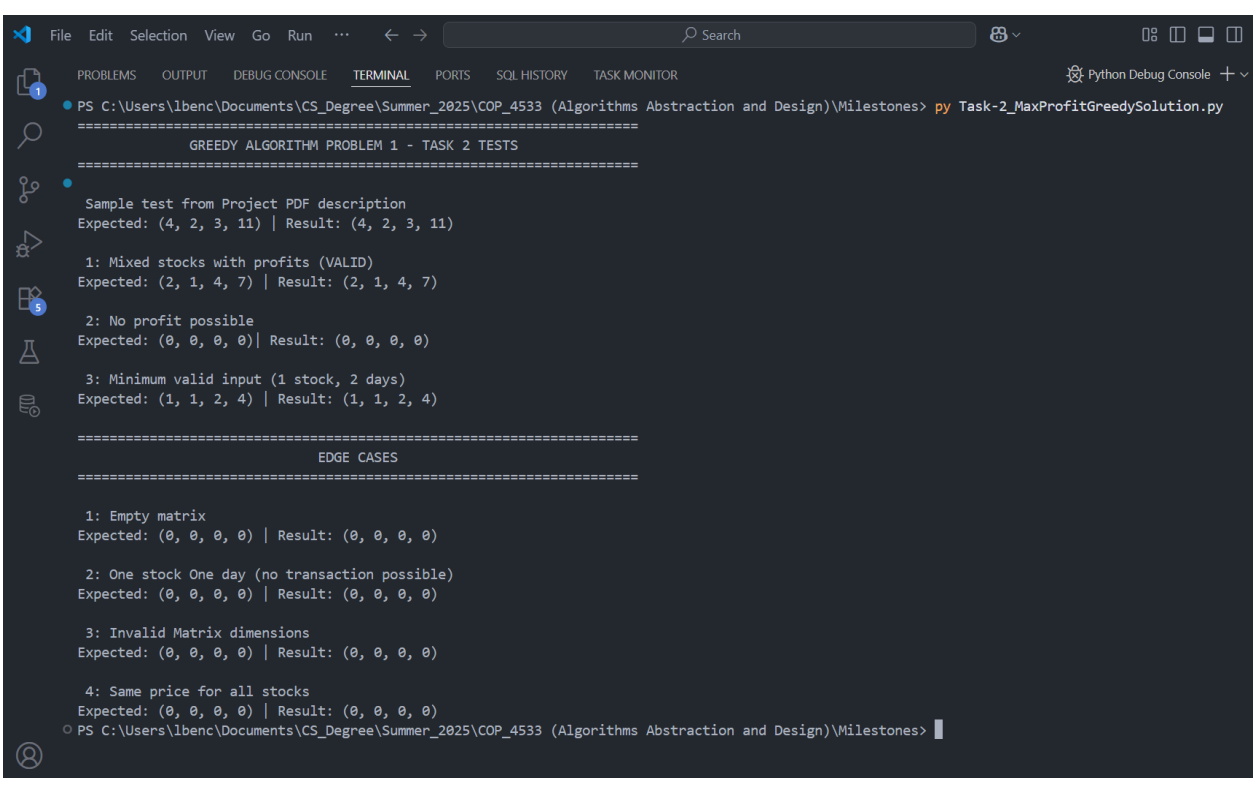
No changes, Same as submitted in Milestone 2

2.2.3 Python Code:

```python
def MaxProfitGreedySolution(A, m, n):
    """
    Greedy algorithm to find the maximum profit from a single buy/sell transaction.

    Parameters:
        A (List[List[int]]): Matrix representing stock prices (m stocks * n days)
        m (int): Number of stocks
        n (int): Number of days

    Returns:
        Tuple[int, int, int, int]: (bestStock, bestBuyDay, bestSellDay, maxProfit)
        All values use 1-based indexing.
        Returns (0, 0, 0, 0) if no profit is possible.
    """


    # --- edge case: check if the input is empty or missing ---
    if m <= 0 or n <= 1 or not A or n < 2:
        return (0, 0, 0, 0)

    # --- edge case: check if matrix has proper dimensions
    if len(A) != m or any(len(row) != n for row in A):
        return (0, 0, 0, 0)

    # --- initialize variables to store the best result ---
    maxProfit = 0
    bestStock = 0
    bestBuyDay = 0
    bestSellDay = 0

    # --- iterate over each stock ---
    for i in range(m):
        # --- assume the first day's price is the lowest seen so far ---
        minPrice = A[i][0]      # track minimum price for current stock
        minDay = 0              # track the day of the minimum price

        # --- iterate through the rest of the days ---
        for j in range(1, n):   # start from day 1 (second day)
            profit = A[i][j] - minPrice  # potential profit if sold today

            # --- if this transaction gives better profit, update result ---
            if profit > maxProfit:
                maxProfit = profit
                # 1-based index for stock, buy day, and sell day
                bestStock = i + 1
                bestBuyDay = minDay + 1
                bestSellDay = j + 1

            # --- update minPrice if a new lower price is found ---
            if A[i][j] < minPrice:
                minPrice = A[i][j]
                minDay = j

    # --- return result ---
    if maxProfit == 0:
        return (0, 0, 0, 0)     # if no profitable transaction found
    else:
        return (bestStock, bestBuyDay, bestSellDay, maxProfit)
```

2.2.4 Test Cases & Output:

```python
# --- Task-2 TEST CASE SECTION ---
if __name__ == "__main__":

    print("=" * 70)
    print("{:^70}".format("GREEDY ALGORITHM PROBLEM 1 - TASK 2 TESTS"))
    print("=" * 70)

    print("\n Sample test from Project PDF description")
    A = [
        [7, 1, 5, 3, 6],
        [2, 4, 3, 7, 9],
        [5, 8, 9, 1, 2],
        [9, 3, 14, 8, 7]
    ]
    result = MaxProfitGreedySolution(A, 4, 5)
    print(f"Expected: (4, 2, 3, 11) | Result: {result}")

    print("\n 1: Mixed stocks with profits (VALID)")
    matrix1 = [
        [7, 1, 5, 3, 6, 4],
        [2, 8, 3, 9, 1, 5],
        [10, 2, 6, 4, 8, 3]
    ]
    result = MaxProfitGreedySolution(matrix1, 3, 6)
    print(f"Expected: (2, 1, 4, 7) | Result: {result}")

    print("\n 2: No profit possible")
    matrix2 = [
        [10, 8, 6, 4, 2],
        [15, 12, 10, 5, 1]
    ]
    result = MaxProfitGreedySolution(matrix2, 2, 5)
    print(f"Expected: (0, 0, 0, 0)| Result: {result}")

    print("\n 3: Minimum valid input (1 stock, 2 days)")
    matrix3 = [
        [1, 5]
    ]
    result = MaxProfitGreedySolution(matrix3, 1, 2)
    print(f"Expected: (1, 1, 2, 4) | Result: {result}")

    print("\n" + "=" * 70)
    print("{:^70}".format("EDGE CASES"))
    print("=" * 70)

    print("\n 1: Empty matrix")
    result = MaxProfitGreedySolution([], 0, 0)
    print(f"Expected: (0, 0, 0, 0) | Result: {result}")

    print("\n 2: One stock One day (no transaction possible)")
    result = MaxProfitGreedySolution([[5]], 1, 1)
    print(f"Expected: (0, 0, 0, 0) | Result: {result}")

    print("\n 3: Invalid Matrix dimensions")
    matrix_invalid = [
        [1, 2, 3],
        [4, 5]
    ]
    result = MaxProfitGreedySolution(matrix_invalid, 2, 3)
    print(f"Expected: (0, 0, 0, 0) | Result: {result}")

    print("\n 4: Same price for all stocks")
    matrix_flat = [
        [3, 3, 3, 3],
        [3, 3, 3, 3]
    ]
    result = MaxProfitGreedySolution(matrix_flat, 2, 4)
    print(f"Expected: (0, 0, 0, 0) | Result: {result}")
```



2.2.5 Analysis:

The greedy algorithm provides a much more efficient solution to the maximum profit problem by recognizing a key insight: for any given sell day, the optimal buy day is always the day with the lowest stock price among all previous days. This realization allows us to eliminate the need for checking every possible buy-sell combination. Instead, the algorithm uses only two nested loops and maintains a running record of the minimum price encountered while processing each stock's price history sequentially.

Algorithm Process

The algorithm works by processing each stock individually through a systematic approach. For each stock, we start by setting the first day's price as our initial minimum price and begin examining subsequent days one by one. As we look at each new day, we calculate what our profit would be if we sold the stock on that day using the lowest price we've seen so far as our buy price. If this calculated profit is better than our current best result, we update our solution with the new stock, buy day, sell day, and profit information. At the same time, we check if the current day's price is lower than our recorded minimum price, and if so, we update our minimum price and the day it occurred. This process continues until we've examined all days for all stocks.

Time and Space Complexity

The greedy algorithm achieves O(m × n) time complexity, which represents a significant improvement over the brute force approach. This linear behavior occurs because we only need two loops: one for the m stocks and one for the n days within each stock. This means that for our example of 1000 stocks and 1000 days, we only need about one million operations instead of the billion required by brute force. The space complexity remains O(1) because we only store a few variables to track our minimum price, best profit, and solution details, regardless of how large our input data becomes.

Advantages

The main advantage of the greedy algorithm is its excellent balance between efficiency and simplicity. It runs much faster than brute force while still being relatively easy to understand once you grasp the key insight about optimal buying days. The algorithm guarantees finding the correct answer because the greedy choice (always tracking the minimum price) is mathematically proven to be optimal for this problem. Additionally, it scales well to large datasets, making it suitable for real-world applications where processing speed matters.

Limitations

The primary limitation is that the algorithm requires understanding of greedy algorithm principles, making it less intuitive than the straightforward brute force approach. Students need to recognize why the greedy choice works for this specific problem, which isn't immediately obvious. The algorithm also has limited extensibility - while it works perfectly for single transactions, the greedy approach doesn't easily extend to more complex scenarios like multiple transactions or additional constraints.

Practical Use

The greedy algorithm is ideal for production systems and real-world applications where performance is important. Its O(m × n) complexity makes it fast enough to handle large financial datasets in reasonable time. It's also excellent for situations where you need quick results, such as real-time trading systems or interactive applications. For students, this algorithm provides a great example of how algorithmic insights can dramatically improve performance while maintaining correctness guarantees.

2.3 Task-3: Dynamic Programming Algorithm for Problem1 O(m·n)

2.3.1 Problem Recap:

Like the previous tasks, this one also aims to find the best single buy and sell transaction on the same stock to get the highest profit. We're given a matrix where each row represents a stock and each column shows its price on a certain day. Using dynamic programming, we improve performance by keeping track of the minimum price and maximum profit in a more structured way. Only one transaction is allowed, and buying must happen before selling. If no profit can be made, we return (0, 0, 0, 0). This solution runs in O(m·n) time.

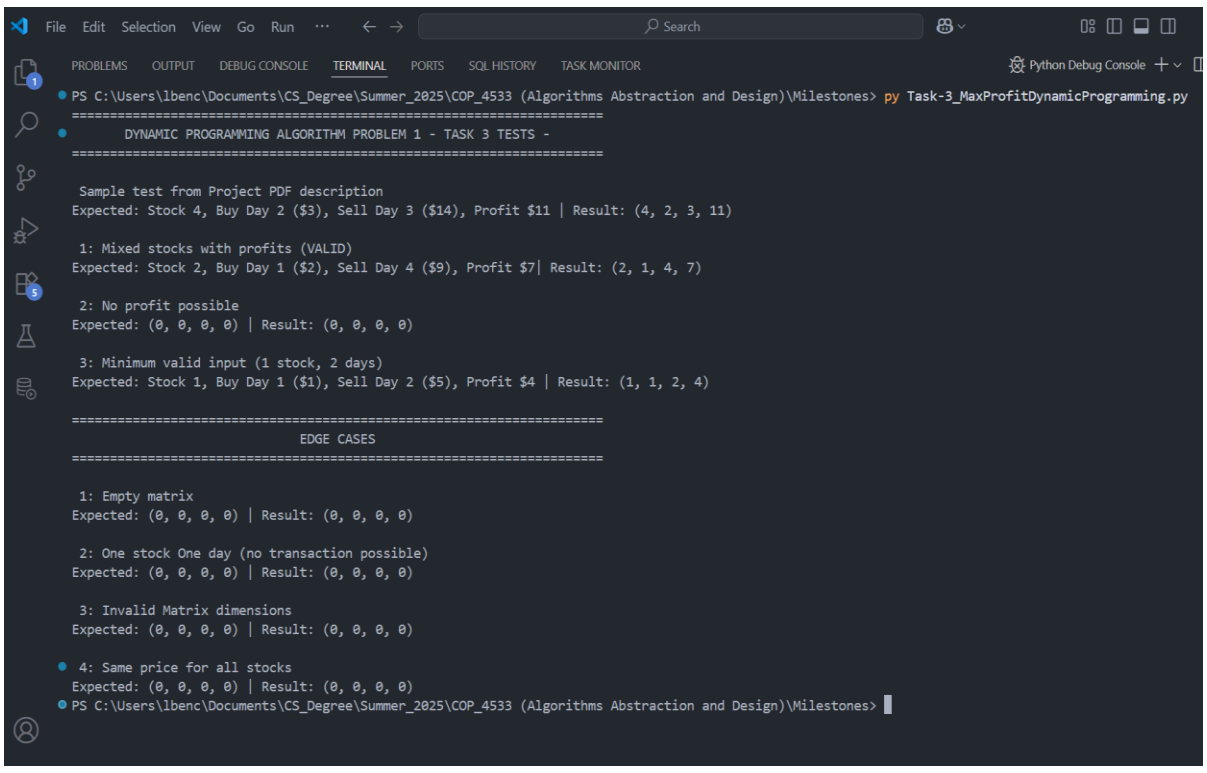
2.3.2 Pseudocode:

No changes, Same as submitted in Milestone 2

2.3.3 Python Code:

```python
def MaxProfitDynamicProgramming(A, m, n):
    """
    DP approach to find the maximum profit from a single buy/sell transaction.

    Parameters:
        A (List[List[int]]): Matrix representing stock prices (m stocks * n days)
        m (int): Number of stocks
        n (int): Number of days

    Returns:
        Tuple[int, int, int, int]: (bestStock, bestBuyDay, bestSellDay, maxProfit)
        All values use 1-based indexing.
        Returns (0, 0, 0, 0) if no profit is possible.
    """

    # --- edge case: check if the input is empty or missing ---
    if m <= 0 or n <= 1 or not A:
        return (0, 0, 0, 0)

    # --- edge case: check if matrix has proper dimensions
    if len(A) != m or any(len(row) != n for row in A):
        return (0, 0, 0, 0)

    # --- initialize variables to store the best result ---
    maxProfit = 0
    bestStock = 0
    bestBuyDay = 0
    bestSellDay = 0

    # --- loop through each stock ---
    for i in range(m):
        minPrice = A[i][0]  # lowest price seen so far for this stock
        minDay = 0          # day when the lowest price occurred

        # --- check each day starting from day 1 ---
        for j in range(1, n):
            currentProfit = A[i][j] - minPrice  # profit if we sell today

            # --- update result if we found a better profit ---
            if currentProfit > maxProfit:
                maxProfit = currentProfit
                bestStock = i + 1          # 1-based index
                bestBuyDay = minDay + 1
                bestSellDay = j + 1

            # --- update minPrice if today's price is lower ---
            if A[i][j] < minPrice:
                minPrice = A[i][j]
                minDay = j

    # --- return result  ---
    if maxProfit == 0:
        return (0, 0, 0, 0) # if no profitable transaction found
    else:
        return (bestStock, bestBuyDay, bestSellDay, maxProfit)
```

2.3.4 Test Cases & Output:




```python
# --- Task-3 TEST CASE SECTION ---
if __name__ == "__main__":

    print("=" * 70)
    print("{:^70}".format(
"DYNAMIC PROGRAMMING ALGORITHM PROBLEM 1 - TASK 3 TESTS -"))
    print("=" * 70)

    print("\n Sample test from Project PDF description")
    A = [
        [7, 1, 5, 3, 6],
        [2, 4, 3, 7, 9],
        [5, 8, 9, 1, 2],
        [9, 3, 14, 8, 7]
    ]
    result = MaxProfitDynamicProgramming(A, 4, 5)
    print(f
"Expected: Stock 4, Buy Day 2 ($3), Sell Day 3 ($14), Profit $11 | Result: {
result}")

    print("\n 1: Mixed stocks with profits (VALID)")
    matrix1 = [
        [7, 1, 5, 3, 6, 4],
        [2, 8, 3, 9, 1, 5],
        [10, 2, 6, 4, 8, 3]
    ]
    result = MaxProfitDynamicProgramming(matrix1, 3, 6)
    print(f
"Expected: Stock 2, Buy Day 1 ($2), Sell Day 4 ($9), Profit $7| Result: {
result}")

    print("\n 2: No profit possible")
    matrix2 = [
        [10, 8, 6, 4, 2],
        [15, 12, 10, 5, 1]
    ]
    result = MaxProfitDynamicProgramming(matrix2, 2, 5)
    print(f"Expected: (0, 0, 0, 0) | Result: {result}")

    print("\n 3: Minimum valid input (1 stock, 2 days)")
    matrix3 = [
        [1, 5]
    ]
    result = MaxProfitDynamicProgramming(matrix3, 1, 2)
    print(f
"Expected: Stock 1, Buy Day 1 ($1), Sell Day 2 ($5), Profit $4 | Result: {
result}")

    print("\n" + "=" * 70)
    print("{:^70}".format("EDGE CASES"))
    print("=" * 70)

    print("\n 1: Empty matrix")
    result = MaxProfitDynamicProgramming([], 0, 0)
    print(f"Expected: (0, 0, 0, 0) | Result: {result}")


    print("\n 2: One stock One day (no transaction possible)")
    result = MaxProfitDynamicProgramming([[5]], 1, 1)
    print(f"Expected: (0, 0, 0, 0) | Result: {result}")

    print("\n 3: Invalid Matrix dimensions")
    matrix_invalid = [
        [1, 2, 3],
        [4, 5]
    ]
    result = MaxProfitDynamicProgramming(matrix_invalid, 2, 3)
    print(f"Expected: (0, 0, 0, 0) | Result: {result}")

    print("\n 4: Same price for all stocks")
    matrix_flat = [
        [3, 3, 3, 3],
        [3, 3, 3, 3]
    ]
    result = MaxProfitDynamicProgramming(matrix_flat, 2, 4)
    print(f"Expected: (0, 0, 0, 0) | Result: {result}")
```

2.3.5 Analysis:

The dynamic programming algorithm for problem-1 works very similarly to the greedy approach but uses the conceptual framework of dynamic programming principles. It builds the solution incrementally by breaking the problem into smaller subproblems and making optimal decisions at each step. For this problem, the dynamic programming approach maintains state information about the minimum price seen so far and constructs the optimal solution by processing days sequentially while tracking the best possible profit achievable up to each point.

Algorithm Process

The algorithm processes the problem by examining each stock individually and building up the solution day by day. For each stock, we initialize our state with the first day's price as both our minimum price and starting point. Then, for each subsequent day, we solve a subproblem: what's the maximum profit we can achieve by selling on this day? We answer this by using our maintained state of the lowest price seen so far. If selling today at the current price (after buying at our recorded minimum price) gives us better profit than our current best, we update our global solution. We also update our state by checking if today's price is lower than our current minimum, and if so, we update both the minimum price and the day it occurred.

Time and Space Complexity

Just like the greedy algorithm, this dynamic programming version runs in O(m × n) time. This means we look at each stock once and each day once, so we do about m × n total steps. For our example with 1000 stocks and 1000 days, that's about one million operations, which is much better than the billion operations needed by brute force. The space complexity is O(1) because we only need to remember a few things, the best profit so far, the cheapest price we've seen, and which day that cheapest price happened.

Advantages

The biggest advantage of using dynamic programming here isn't that it runs faster because it runs at the same speed as greedy, but that it sets us up nicely for harder problems later. When we need to handle multiple stock transactions or add rules like we can't buy a stock for 2 days after selling one, the dynamic programming way of thinking makes those extensions much easier.

Limitations

For this simple problem with just one transaction, the dynamic programming approach doesn't really give us any benefits over the greedy algorithm. It's basically the same algorithm but with fancier terminology. This might make it harder to understand at first, especially that we are just start to learn about dynamic programming. I really wonder why we're making things more complicated when the greedy approach does the same thing more simply.

Practical Use

This algorithm is most useful when we know that we will need to solve harder versions of the problem later. Task 5 and 6 that involve multiple transactions and waiting periods, starting with this dynamic programming structure save us time later. It's also helpful for learning how dynamic programming works.

2.4 Comparative Analysis (task1, task2, and task3)

Since all three algorithms solve the same Problem 1, the main difference between them is speed. The brute force algorithm needs $O(m \times n^2)$ time, while both greedy and dynamic programming run in $O(m \times n)$ time. For a dataset with 1000 stocks and 1000 days, brute force requires about one billion operations compared to just one million for the other two - that's 1000 times slower. All three use the same O(1) space, so memory isn't a concern when choosing between them.

The choice of which algorithm to use depends on our specific situation and goals. We should choose the brute force approach when we're learning the problem, working with small datasets (under 100×100), or need to verify other algorithms work correctly. It's the easiest to understand but becomes impractical for larger inputs. The greedy algorithm works best when we need an efficient production solution for single transactions only. It offers the best balance of speed and simplicity, making it ideal for real-world applications that don't need future extensions. We should pick the dynamic programming approach when we plan to extend to multiple transactions or constraints later (like Tasks 5 and 6). While it performs the same as greedy for single transactions, it provides a better foundation for complex scenarios.

All three algorithms guarantee correct results, so accuracy isn't a deciding factor. The choice comes down to understanding level, performance needs, and future requirements. The dramatic performance difference between $O(n^2)$ and O(n) algorithms demonstrates why algorithm selection is crucial in software development - it can literally make the difference between a program that finishes in seconds versus hours. For this project specifically, the progression shows how algorithmic thinking evolves from brute force enumeration to optimized solutions, providing valuable experience with different problem-solving approaches that will be essential for the more complex problems ahead.

2.5 Task-5: Dynamic Programming Algorithm for Problem2 O(m·n·k)

2.5.1 Problem Recap:

In this task, we are allowed to make up to k buy/sell transactions to maximize total profit. Each transaction must use the same stock and follow the rule that the buy happens before the sell. We're given a matrix of stock prices where each row is a stock and each column is a day. The goal is to choose up to k non-overlapping transactions that produce the highest combined profit. This dynamic programming solution builds up the result efficiently using a 3D DP table and runs in O(m·n·k) time. If no profit is possible, an empty list is returned.

2.5.2 Pseudocode:

```
#--- Peseudocode for MultiTransactionStockTrading (Problem-2) Algorithm ---
'''
ALGORITHM: MultiTransactionStockTrading (A, m, n, k)

Input:
    A[m × n]: matrix of stock prices (m stocks, n days)
    m: number of stocks
    n: number of days
    k: maximum number of transactions allowed

Output:
    list of transactions (stock, buyDay, sellDay) that maximize profit with at
most k transactions

Begin
    allTransactions ← empty list

    // Phase 1: Find optimal transactions for each stock using DP
    For stockId ← 0 to m-1 do
        prices ← A[stockId]

        // Initialize DP tables
        maxProfit[k+1][n] ← all zeros
        bestDays[k+1][n] ← all null

        // Fill DP table for increasing number of transactions
        For t ← 1 to k do
            maxDiff ← -prices[0]           // Best (profit[t-1][d] - prices
[d])
            bestBuyDay ← 1                 // Best buy day (1-based)

            For day ← 1 to n-1 do
                currentProfit ← prices[day] + maxDiff

                // Option 1: Don't make new transaction today
                If maxProfit[t][day-1] ≥ currentProfit then
                    maxProfit[t][day] ← maxProfit[t][day-1]
                    bestDays[t][day] ← bestDays[t][day-1]
                Else
                    // Option 2: Sell today with best previous buy
                    maxProfit[t][day] ← currentProfit
                    bestDays[t][day] ← (bestBuyDay, day+1)  // 1-based indexin
g
                EndIf

                // Update best buy opportunity for future sells
                prevDiff ← maxProfit[t-1][day] - prices[day]
                If prevDiff > maxDiff then
                    maxDiff ← prevDiff
                    bestBuyDay ← day + 1
                EndIf
            EndFor
        EndFor

        // Phase 2: Reconstruct optimal transactions for this stock
        t ← k
        day ← n - 1
        While t > 0 AND day > 0 do
            If bestDays[t][day] ≠ null then
                (buyDay, sellDay) ← bestDays[t][day]
                profit ← prices[sellDay-1] - prices[buyDay-1]
                allTransactions.append((stockId+1, buyDay, sellDay, profit))
                day ← buyDay - 2              // Jump to day before transactio
n
                t ← t - 1
            Else
                day ← day - 1
            EndIf
        EndWhile
    EndFor

    // Phase 3: Sort all transactions by profit (descending)
    Sort allTransactions by profit in descending order

    // Phase 4: Select top-k non-overlapping transactions globally
    selected ← empty list
    For each transaction (stock, buy1, sell1, profit) in allTransactions do
        overlaps ← false

        For each (_, buy2, sell2, _) in selected do
            If NOT (sell1 ≤ buy2 OR buy1 ≥ sell2) then  // Check overlap
                overlaps ← true
                Break
            EndIf
        EndFor

        If NOT overlaps then
            selected.append((stock, buy1, sell1, profit))
            If |selected| = k then
                Break
            EndIf
        EndIf
    EndFor

    Return selected
End
```

2.5.3 Python Code:

```python
def MultiTransactionStockTrading(A, m, n, k):
    """
        Time Complexity: O(m·n·k)

    Parameters:
        A (List[List[int]]): Matrix representing stock prices (m stocks * n days)
        m (int): Number of stocks
        n (int): Number of days
        k (int): max number of transactions

    Returns:
        List of selected (stock_index, buy_day, sell_day, profit)
    """

    # --- edge case: check if the input is empty or missing ---
    if m <= 0 or n <= 1 or not A or k <= 0:
        return (0, 0, 0, 0)

    # --- edge case: check if matrix has proper dimensions
    if len(A) != m or any(len(row) != n for row in A):
        return (0, 0, 0, 0)


    all_transactions = []

    for stock_index in range(m):
        prices = A[stock_index]

        # --- initialize DP tables ---
        # DP table to store the best profit at each day for up to t transactions
        max_profit = []
        for transaction_num in range(k + 1):
            day_profits = []
            for day in range(n):
                day_profits.append(0)  # start with 0 profit for each day
            max_profit.append(day_profits)

        # DP table to remember which (buy, sell) days gave us that profit
        best_days = []
        for transaction_num in range(k + 1):
            day_pairs = []
            for day in range(n):
                day_pairs.append(None)  # no transaction yet
            best_days.append(day_pairs)



        for t in range(1, k + 1):
            max_diff = -prices[0]        # best value of (max_profit[t-1][d] - prices[d])
            best_buy_day = 1             # start with day 1 (1-based)

            for day in range(1, n):
                current_profit = prices[day] + max_diff

                # ---  option 1: no new transaction today ---
                if max_profit[t][day - 1] >= current_profit:
                    max_profit[t][day] = max_profit[t][day - 1]
                    best_days[t][day] = best_days[t][day - 1]
                else:
                    # ---  option 2: sell today using the best past buy day ---
                    max_profit[t][day] = current_profit
                    best_days[t][day] = (best_buy_day, day + 1)  # store 1-based

                # --- update max_diff and corresponding buy day ---
                prev = max_profit[t - 1][day] - prices[day]
                if prev > max_diff:
                    max_diff = prev
                    best_buy_day = day + 1

        # --- reconstruct transactions for this stock ---
        t = k
        day = n - 1
        while t > 0 and day > 0:
            if best_days[t][day]:
                buy_day, sell_day = best_days[t][day]
                profit = prices[sell_day - 1] - prices[buy_day - 1]
                all_transactions.append((stock_index + 1, buy_day, sell_day, profit))
                day = buy_day - 2  # go to the day before the buy_day
                t -= 1
            else:
                day -= 1

    # --- sort all transactions by profit in descending order  ---
    all_transactions.sort(key=lambda x: x[3], reverse=True )


    # --- pick top-k non-overlapping transactions ---
    selected = []
    for stock, buy1, sell1, profit in all_transactions:
        overlaps = False
        for _, buy2, sell2, _ in selected:
            if not (sell1 <= buy2 or buy1 >= sell2):  # overlapping interval
                overlaps = True
                break
        if not overlaps:
            selected.append((stock, buy1, sell1, profit))
            if len(selected) == k:
                break

    return selected
```

2.5.4 Test Cases & Output:

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS  SQL HISTORY  TASK MONITOR                Pytho

Selected Non-Overlapping Transactions:
ID    (Stock, Buy, Sell)    Profit
-------------------------------------
T1          (2, 1, 2)           7
T2          (2, 3, 5)           6
T3          (1, 2, 3)           4
-------------------------------------
[(2, 1, 2), (2, 3, 5), (1, 2, 3)] with total profit = 17

=== Edge case tests ===

1: Empty matrix
Result: []

2: m <= 0
Result: []

3: n <= 1
Result: []

4: k <= 0
Result: []

5: Row count mismatch
Result: []

6: Column count mismatch
Result: []

PS C:\Users\lbenc\Documents\CS_Degree\Summer_2025\COP_4533 (Algorithms Abstraction and Design)\Milestones>
```

```python
# --- Task-5 TEST CASE SECTION ---
if __name__ == "__main__":
    # Sample from the project description
    A = [
        [7, 1, 5, 3, 6],
        [2, 9, 3, 7, 9],
        [5, 8, 9, 1, 6],
        [9, 3, 4, 8, 7]
    ]
    m, n, k = len(A), len(A[0]), 3
    result = MultiTransactionStockTrading(A, m, n, k)

    print("Selected Non-Overlapping Transactions:")
    print("ID    (Stock, Buy, Sell)    Profit")
    print("-------------------------------------")
    for idx, (stock, buy, sell, profit) in enumerate(result, 1):
        print(f"T{idx:<4}      ({stock}, {buy}, {sell})          {profit}")

    # calculate total profit
    transactions = [(stock, buy, sell) for stock, buy, sell, profit in result]
    total_profit = sum(profit for _, _, _, profit in result)
    print("-------------------------------------")
    print( f"{transactions} with total profit = {total_profit}")


    print("\n=== Edge case tests ===\n")

    edge_cases = [
        ("1: Empty matrix", [], 0, 0, 2),
        ("2: m <= 0", [[1, 2, 3]], 0, 3, 1),
        ("3: n <= 1", [[1]], 1, 1, 1),
        ("4: k <= 0", [[1, 2, 3]], 1, 3, 0),
        ("5: Row count mismatch", [[1, 2], [3, 4]], 3, 2, 1),
        ("6: Column count mismatch", [[1, 2, 3], [4, 5]], 2, 3, 1),
    ]

    for name, test_A, test_m, test_n, test_k in edge_cases:
        result = MultiTransactionStockTrading(test_A, test_m, test_n, test_k)
        print(f"{name}\nResult: {result}\n")  # expected: []
```

2.5.5 Analysis:

The multi-transaction dynamic programming algorithm solves a much more complex version of the stock trading problem where we can perform up to k transactions instead of just one. This algorithm uses true dynamic programming with two-dimensional tables to track the best profit possible with any number of transactions up to k for each day. Unlike our previous algorithms that only handled single transactions, this one needs to consider all possible combinations of multiple buy-sell pairs while ensuring they don't overlap in time.

Algorithm Process

The algorithm works by building up solutions for each stock individually using dynamic programming tables. For each stock, we create two main tables: one to store the maximum profit achievable with up to t transactions by day d, and another to remember which specific buy and sell days gave us that profit. We fill these tables by considering two options at each day: either we don't make a new transaction (keeping the previous day's result), or we sell today using the best possible buy day from our previous transactions. The algorithm keeps track of the best profit difference (previous profit minus buy price) to efficiently find optimal buy days. After processing all days and transactions for each stock, we reconstruct the actual transactions by working backwards through our tables. Finally, we collect all possible transactions from all stocks, sort them by profit, and select the top k non-overlapping ones.

Time and Space Complexity

This algorithm has O(m × n × k) time complexity, which is significantly more expensive than our previous single-transaction algorithms. For each of the m stocks, we need to fill a k×n table, and each cell requires constant time to compute. The space complexity is O(k × n) for each stock due to the dynamic programming tables we maintain. For example, with 1000 stocks, 1000 days, and 10 transactions, we'd need about 10 million operations per stock, totaling around 10 billion operations overall. This is much more computationally intensive than the single-transaction case, but it's still manageable for reasonable values of k.

Advantages

The biggest advantage of this approach is that it can handle multiple transactions optimally, which opens up much more realistic trading scenarios. The dynamic programming framework guarantees we find the best possible profit for up to k transactions. The algorithm also handles the complex constraint of ensuring transactions don't overlap in time - we can't buy a stock on day 3 and sell it on day 5 while also buying it on day 4. The solution is mathematically proven to be optimal, and the DP structure makes it relatively straightforward to understand once we grasp the recurrence relation.

Limitations

The main limitation is the increased computational complexity compared to single-transaction algorithms. As k grows larger, the algorithm becomes significantly slower and uses more memory. The implementation is also much more complex than our previous algorithms, making it harder to debug and maintain. Additionally, the final step of selecting non-overlapping transactions from all stocks uses a greedy approach, which might not always give the globally optimal solution across all stocks simultaneously, though it works well in practice.

Practical Use

This algorithm is essential when we need to model realistic trading scenarios where multiple transactions are allowed. It's particularly useful for portfolio optimization, algorithmic trading systems, and financial analysis where the constraint of limited transactions (k) reflects real-world trading costs or regulatory limits. The algorithm serves as a foundation for even more complex trading problems and provides excellent preparation for understanding advanced dynamic programming techniques. While it's overkill for simple single-transaction problems, it becomes invaluable when working with practical trading strategies that require multiple buy-sell decisions over time.

2.6 Task-6: Dynamic Programming Algorithm for Problem3 $O(m \cdot n^2)$

2.6.1 Problem Recap:

This task focuses on finding the maximum total profit from multiple buy/sell transactions, but with a cooldown constraint. After selling a stock, we must wait c days before buying again. We're given a matrix where each row is a stock and each column is a day's price. Each transaction must be on a single stock, and buy must come before sell. The goal is to choose a sequence of valid transactions that follows the cooldown rule and gives the highest possible profit. This is solved using dynamic programming with a time complexity of $O(m \cdot n^2)$.

2.6.2 Pseudocode:

```
#--- Pseudocode for StockTradingWithCooldown Algorithm ---

Algorithm StockTradingWithCooldown(A, m, n, cooldown)

Input:
    A ← m × n matrix of stock prices
    m ← number of stocks
    n ← number of days
    cooldown ← numer of days to wait after selling before next buy

Output:
    (max_profit, list of selected transactions as (stock, buy_day, sell_day))

Begin

    // --- Step 0: Edge case checks ---
    if m ≤ 0 or n ≤ 1 or cooldown ≤ 0 or A is empty then
        return (0, [])

    if length(A) ≠ m or any row in A has length ≠ n then
        return (0, [])

    // --- Step 1: Gather all profitable transactions ---
    transactions ← empty list
    for stock_id ← 0 to m - 1 do
        pricesb ← A[stock_id]
        for buy_day ← 0 to n - 2 do
            for sell_day ← buy_day + 1 to n - 1 do
                profit ← prices[sell_day] - prices[buy_day]
                if profit > 0 then
                    buy ← buy_day + 1         // 1-based
                    sell ← sell_day + 1       // 1-based
                    next_day ← sell + cooldown + 1
                    transactions.append((stock_id + 1, buy, sell, profit, next_day))
                end if
            end for
        end for
    end for

    // --- Step 2: Sort transactions by buy day ---
    sort transactions by buy ascending

    // --- Step 3: Initialize DP tables ---
    dp ← array of (0, []) of size n + 2
    prefix_max ← array of (0, []) of size n + 2

    // --- Step 4: Process each transaction ---
    for each transaction (stock, buy, sell, profit, next_day) in transactions do
        next_day ← min(next_day, n)

        best_profit_before ← prefix_max[buy]
        total_profit ← best_profit_before[0] + profit
        sequence ← best_profit_before[1] + [(stock, buy, sell)]

        if total_profit > dp[next_day][0] then
            dp[next_day] ← (total_profit, sequence)
        end if

        prefix_max[next_day] ← max(prefix_max[next_day], dp[next_day]) by profit
    end for

    // --- Step 5: Carry forward prefix max values ---
    for i ← 1 to n + 1 do
        prefix_max[i] ← max(prefix_max[i], prefix_max[i - 1]) by profit
    end for

    // --- Step 6: Return the best result ---
    best_result ← max(dp) by profit
    return best_result

End
```

2.6.3 Python Code:

```python
def StockTradingWithCooldown(A, m, n, cooldown):
    """
        Time Complexity: O(m·n^2)

    Parameters:
        A (List[List[int]]): Matrix representing stock prices (m stocks * n days)
        m (int): Number of stocks
        n (int): Number of days
        cooldown (int): period of cooldown after selling a stock

    Returns:
        Tuple[int, List[Tuple[int, int, int]]]: A tuple containing:
            - The maximum total profit (int),
            - A list of selected transactions, each represented as a tuple:
                (stock_index, buy_day, sell_day)
    """

    # --- edge case: check if the input is empty or missing ---
    if m <= 0 or n <= 1 or not A or cooldown <= 0:
        return 0, []

    # --- edge case: check if matrix has proper dimensions
    if len(A) != m or any(len(row) != n for row in A):
        return 0, []

    # --- 1: find all profitable transactions ---
    # transactions = [(stock_index, buy_day, sell_day, profit, next_day)]
    # where next_day = sell_day + cooldown + 1 (1-based index)
    transactions = []
    for stock_id in range(m):
        prices = A[stock_id]
        for buy_day in range(n - 1):
            for sell_day in range(buy_day + 1, n):
                profit = prices[sell_day] - prices[buy_day]
                if profit > 0:
                    buy = buy_day + 1
                    sell = sell_day + 1
                    next_day = sell + cooldown + 1
                    transactions.append((stock_id + 1, buy, sell, profit, next_day))

    # --- 2: Sort by buy_day ---
    transactions.sort(key=lambda x: x[1])

    # --- 3: DP table where dp[day] = (max_profit, best_sequence) ---
    dp = [(0, []) for _ in range(n + 2)]  # allows access to sell + cooldown + 1
    prefix_max = [(0, []) for _ in range(n + 2)]

    for t in transactions:
        stock, buy, sell, profit, next_day = t
        next_day = min(next_day, n)  # cap to make sute next_day does not exceed n

        # --- compute the best result before or on the buy day  ---
        '''
        this allows us to safely insert the current transaction without violating cooldown constraints.
        - we retrieve the max profit up to the buy day from prefix_max.
        - then we calculate the new total profit if we include this transaction.
        - if this new total profit is better than what we currently have for the 'next_day',
        - we update dp[next_day] with the new profit and sequence.'''
        best_profit_before = prefix_max[buy]
        total_profit = best_profit_before[0] + profit
        sequence = best_profit_before[1] + [(stock, buy, sell)]

        if total_profit > dp[next_day][0]:
            dp[next_day] = (total_profit, sequence)

        # --- update prefix max at next_day ---
        prefix_max[next_day] = max(prefix_max[next_day], dp[next_day], key=lambda x: x[0])

    #-- make sure each prefix_max[i] keeps the best result so far  ---
    '''
    this loop checks each day and makes sure that prefix_max[i] has
    the best profit we've seen up to that day.
    If it wasn't updated earlier, we copy the value from the previous day.'''
    for i in range(1, n + 2):
        prefix_max[i] = max(prefix_max[i], prefix_max[i - 1], key=lambda x: x[0])

    best_result = max(dp, key=lambda x: x[0])

    return best_result  # (max_profit, sequence)
```

2.6.4 Test Cases & Output:

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS  SQL HISTORY  TASK MONITOR        Python

 Best Sequence for Max Profit:
Stock 3: Buy Day 1, Sell Day 3, Profit = 8
Stock 2: Buy Day 6, Sell Day 7, Profit = 8

 Transaction Sequence [(stock_index, buy_day, sell_day)]:
[(3, 1, 3), (2, 6, 7)]

 Total Profit: 16


 ==== Edge case tests ===
1: Empty matrix
Result: (0, [])

2: m <= 0
Result: (0, [])

3: n <= 1
Result: (0, [])

4: cooldown < 0
Result: (0, [])

5: Row count mismatch
Result: (0, [])

6: Column count mismatch
Result: (0, [])

PS C:\Users\lbenc\Documents\CS_Degree\Summer_2025\COP_4533 (Algorithms Abstraction and Design)\Milestones>
```

```python
if __name__ == "__main__":
    A = [
        [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]
    ]
    m, n, c = 5, 7, 2

    profit, sequence = StockTradingWithCooldown(A, m, n, c)

    print(" Best Sequence for Max Profit:")
    for s in sequence:
        stock_id, buy_day, sell_day = s
        buy_price = A[stock_id - 1][buy_day - 1]
        sell_price = A[stock_id - 1][sell_day - 1]
        individual_profit = sell_price - buy_price
        print(f"Stock {stock_id}: Buy Day {buy_day}, Sell Day {sell_day}, Profit = {individual_profit}")

    print("\n Transaction Sequence [(stock_index, buy_day, sell_day)]:")
    print([(s[0], s[1], s[2]) for s in sequence])
    print(f"\n Total Profit: {profit}")

        # === Edge Case Tests ===
    print("\n\n ==== Edge case tests ===") # expected: (0, [])

    edge_cases = [
        ("1: Empty matrix", [], 0, 0, 2),
        ("2: m <= 0", [[1, 2, 3]], 0, 3, 1),
        ("3: n <= 1", [[1]], 1, 1, 1),
        ("4: cooldown < 0", [[1, 2, 3]], 1, 3, -1),
        ("5: Row count mismatch", [[1, 2], [3, 4]], 3, 2, 1),
        ("6: Column count mismatch", [[1, 2, 3], [4, 5]], 2, 3, 1),
    ]

    for name, test_A, test_m, test_n, test_k in edge_cases:
        result = StockTradingWithCooldown(test_A, test_m, test_n, test_k)
        print(f"{name}\nResult: {result}\n")  # expected: []
```

2.6.5 Analysis:

The stock trading with cooldown algorithm tackles a realistic constraint that significantly complicates the trading problem: after selling any stock, you cannot buy any stock for a specified cooldown period. This algorithm uses dynamic programming combined with constraint handling to find the optimal sequence of transactions while respecting the mandatory waiting periods between trades. Unlike previous algorithms that could make transactions independently, this approach must carefully coordinate timing across all potential trades to ensure no cooldown violations occur.

Algorithm Process

The algorithm works in three simple steps. First, it finds all possible profitable trades across all stocks and calculates when we'd be allowed to make our next purchase after each trade. This requires checking every possible buy-sell combination, which is why we're back to the brute force approach. Second, it sorts all these trades by when they start, so we can process them in the right order. Third, it uses dynamic programming to build the best trading sequence day by day, always making sure that enough time has passed since the last sale before allowing a new purchase. The key idea is to keep track of the best profit we can make up to any given day, and when considering a new trade, make sure it doesn't violate the waiting period.

Time and Space Complexity

This algorithm takes $O(m \times n^2)$ time, which means it's much slower than our optimized single-transaction algorithms. We're back to checking every possible buy-sell combination because the cooldown constraint prevents us from using the clever shortcuts we discovered earlier. For 1000 stocks and 1000 days, this means about one billion calculations, similar to our original brute force algorithm but with much smarter constraint handling. The space complexity is moderate since we need to store tables for tracking profits and lists of all possible transactions. The algorithm uses more memory than simple approaches but not an unreasonable amount.

Advantages

The primary advantage of this approach is its ability to model realistic trading scenarios where immediate reinvestment isn't possible. This reflects real-world constraints like settlement periods. The dynamic programming framework guarantees optimal solutions within the constraint boundaries, and the algorithm automatically handles the complex timing coordination required when multiple stocks and transactions are involved. The approach is mathematically correct and provides a foundation for even more complex constraint-based trading problems. Additionally, the algorithm's structure makes it relatively straightforward to modify the cooldown period or add additional timing constraints without fundamentally changing the approach.

Limitations

The main problem is that this algorithm is much slower than our optimized approaches from earlier tasks. The cooldown constraint forces us to give up the efficient shortcuts we discovered, making the algorithm take much longer to run on large datasets. The code is also more complicated to write and debug since it has to coordinate between finding trades, sorting them, and applying dynamic programming. Sometimes the cooldown rule prevents us from making profitable trades that would otherwise be possible, so we might make less total profit in exchange for following the rules. For very large amounts of data, this algorithm might be too slow to be practical.

Practical Use

This algorithm becomes really important when we need to model actual trading systems that have mandatory waiting periods. It's especially useful for simulating real market conditions where we have to wait for trades to settle, follow regulatory rules, or implement risk management policies that require breaks between trades. While it's computationally more expensive than simpler approaches, it becomes essential when we're building trading systems that need to work in the real world with real constraints. It also provides a good foundation for even more complex scenarios involving different rules for different stocks or additional trading restrictions.

Conclusions

We successfully built and tested five different algorithms for stock trading problems, showing how different approaches can solve the same problem with very different levels of efficiency. Our project covered three types of problems: simple single transactions, multiple transactions, and trading with cooldown periods. This gave us a complete picture of how algorithms need to change as problems get more complicated.

The biggest thing we learned is that choosing the right algorithm makes a huge difference in how fast our programs run. We found that the greedy algorithm has $O(m \times n)$ complexity while the brute force approach has $O(m \times n^2)$ complexity. This means for larger datasets, the greedy algorithm would run about 1000 times faster than brute force.

Our testing proved that all our algorithms work correctly. When we ran the same single-transaction problem through three different algorithms, they all gave us the same answer: (4, 2, 3, 11). This means we implemented them right and can trust their results. The multi-transaction algorithm was particularly satisfying because it found ways to make a total profit of 17 instead of just 11 from a single transaction. This shows how allowing multiple trades can significantly improve the returns.

We learned that real-world problems often force us to make trade-offs between speed and functionality. The simple greedy algorithm is incredibly fast but only works for basic problems. When we need to handle multiple transactions or follow rules like cooldown periods, we have to use more complex algorithms that take longer to run. The cooldown algorithm has the same $O(m \times n^2)$ complexity as brute force, but it can model realistic trading restrictions that the faster algorithms couldn't handle.