Milestone 3: Algorithm Design

# 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:**

# **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.

# **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

# **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·n2).

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 j1 and selling it later on day j2, where j1 < j2.
* The transaction must be on the same stock (row).
* The result should be a tuple: (i, j1, j2, profit) where:
* i: index of the chosen stock (1-based index)
* j1: day to buy
* j2: day to sell
* profit = A[i][j2] - A[i][j1]

**Pseudocode:**

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 (j1, j2) 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 (j₁ < j₂)
* Each stock is evaluated independently
* Return (0, 0, 0, 0) if no profit is possible

**Pseudocode:**

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 j1 and selling it later on day j2, where j1 < j2.
* The transaction must be on the same stock (row).
* The result should be a tuple: (i, j1, j2, profit) where:
* i: index of the chosen stock (1-based index)
* j1: day to buy
* j2: day to sell
* profit = A[i][j2] - A[i][j1]

**Pseudocode:**

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 to 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 j₁ and selling it on day j₂, where j₁ < j₂
* 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: [(i₁, j₁, j₂), (i₂, j₃, j₄), ...] representing optimal transactions

**Pseudocode:**

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 transactions

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·n2)

The goal here is to maximize total profit from multiple stock transactions with a cooldown period c. After selling a stock on day j2, the next allowed buy can only happen on day j2 + 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)

**Pseudocode:**

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] then

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][currentDay] 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

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

**Individual submission:**

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

## 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.

## Programming Language Used

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

## 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·n²)
* 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·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·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·n·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·n²)

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·n2).

#### 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·n²)

#### 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 transaction.

    Parameters:

    A (List[List[int]]): Matrix representing stock prices (m stocks x 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.

    """

    # --- 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

#### 2.1.5 Analysis

### 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:

#### 2.2.4 Test Cases & Output:

#### 2.2.5 Analysis:

### 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:

#### 2.3.4 Test Cases & Output:

#### 2.3.5 Analysis:

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

#### 2.4.1 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.

#### 2.4.2 Pseudocode:

No changes, Same as submitted in Milestone 2

#### 2.4.3 Python Code:

#### 2.4.4 Test Cases & Output:

#### 2.4.5 Analysis:

### 2.5 Task-6: Dynamic Programming Algorithm for Problem3 O(m·n2)

#### 2.5.1 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·n²).

#### 2.5.2 Pseudocode:

No changes, Same as submitted in Milestone 2

#### 2.5.3 Python Code:

#### 2.5.4 Test Cases & Output:

#### 2.5.5 Analysis: