Table of Contents

Milestone 2: Algorithm Design

1. Milestone 1: Understanding the problems

GitHub Repository Link: https://github.com/Angellsh/COP4533-Group-13.git

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

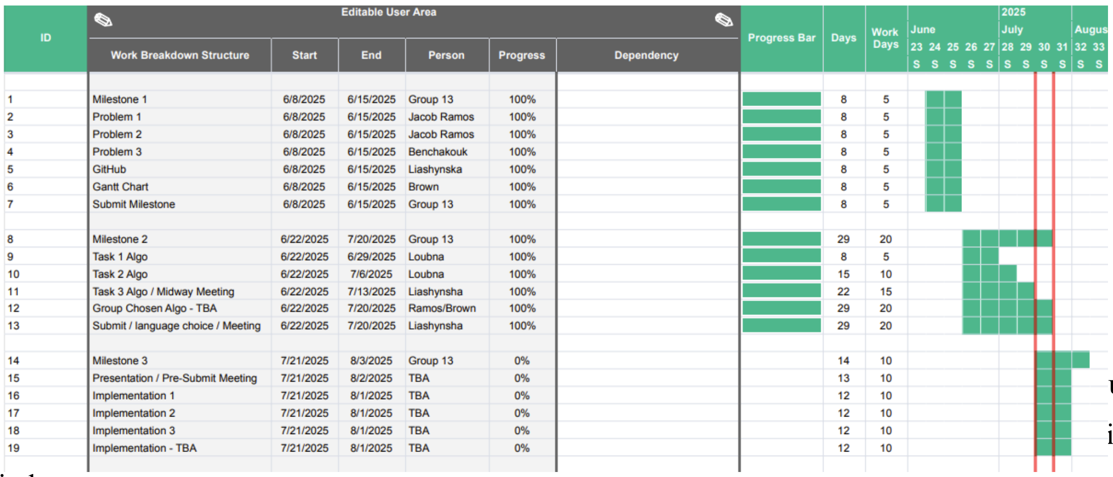
Member Roles:

- **Jacob Ramos**: Task 6
- **Loubna Benchakouk**: Leaving group. Allowed us to keep their work on Task 1 and 2
- **Anhelina Liashynska**: Task 3 and 5
- **Dani Brown**: Gantt Chart, Task 7

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

Programming Language: Python

Project Gantt Chart:



ID	Editable User Area						Progress Bar	Days	Work Days	June					2025 July				August	
	Work Breakdown Structure	Start	End	Person	Progress	Dependency				23 S	24 S	25 S	26 S	27 S	28 S	29 S	30 S	31 S	32 S	33 S
1	Milestone 1	6/8/2025	6/15/2025	Group 13	100%		🟩	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	100%		🟩	8	5		🟩	🟩								
8	Milestone 2	6/22/2025	7/20/2025	Group 13	100%		🟩	29	20				🟩	🟩	🟩	🟩	🟩			
9	Task 1 Algo	6/22/2025	6/29/2025	Loubna	100%		🟩	8	5				🟩	🟩						
10	Task 2 Algo	6/22/2025	7/6/2025	Loubna	100%		🟩	15	10				🟩	🟩	🟩					
11	Task 3 Algo / Midway Meeting	6/22/2025	7/13/2025	Liashynsha	100%		🟩	22	15				🟩	🟩	🟩	🟩				
12	Group Chosen Algo - TBA	6/22/2025	7/20/2025	Ramos/Brown	100%		🟩	29	20				🟩	🟩	🟩	🟩	🟩			
13	Submit / language choice / Meeting	6/22/2025	7/20/2025	Liashynsha	100%		🟩	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								🟩	🟩		

umn will

ing 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

2. Milestone 2: Algorithm Design

<u>Language decision</u>:
We have decided to utilize Python for our implementation.

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 = $A[i][j_2]$ - $A[i][j_1]$

Pseudocode:

```
Algorithm MaxProfitBruteForce(A, m, n)
Input: matrix A(m × n) representing stock prices
Output: tuple (i, j₁, j₂, profit) representing the best stock and days to buy/sell and max profit

maxProfit ← 0
bestStock ← 0
bestBuyDay ← 0
bestSellDay ← 0

for i ← 0 to m − 1 do
    for j₁ ← 0 to n − 2 do
        for j₂ ← j₁ + 1 to n − 1 do
            profit ← A[i][j₂] − A[i][j₁]
            if profit > maxProfit then
                maxProfit ← profit
                bestStock ← i + 1
                bestBuyDay ← j₁ + 1
                bestSellDay ← j₂ + 1

if maxProfit = 0 then
    return (0, 0, 0, 0)      // no profitable transaction
else
    return (bestStock, bestBuyDay, bestSellDay, maxProfit)
```

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 forProblem1 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

Pseudocode:

```
Algorithm MaxProfitGreedy(A, m, n)
Input: matrix A(m × n) representing stock prices
Output: tuple (i, $j_1$, $j_2$, profit) representing the best stock and days to buy/sell and max profit

  maxProfit ← 0
  bestStock ← 0
  bestBuyDay ← 0
  bestSellDay ← 0

  for i ← 0 to m − 1 do
    minPrice ← A[i][0]
    minDay ← 0

    for j ← 1 to n − 1 do
      if A[i][j] − minPrice > maxProfit  then
        maxProfit ← A[i][j] − minPrice
        bestStock ← i + 1
        bestBuyDay ← minDay + 1
        bestSellDay ← j + 1

      if A[i][j] < minPrice then
        minPrice ← A[i][j]
        minDay ← j

  if maxProfit = 0 then
    return (0, 0, 0, 0)
  else
    return (bestStock, bestBuyDay, bestSellDay, maxProfit)
```

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).

Time Complexity: $O(m \cdot n)$

The Algorithm iterates over each of the (m) stocks
For each stock, it performs a single pass over (n) daily prices
In each pass, comparisons and updates are done in constant time
⇨ Total = $O(m \cdot n)$

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

Assumptions & variable definitions:

- input stocks is a 2D list of shape $m \times n$

- the transaction must obey buy_day < sell_day.

Definitions:

- m – number of stocks
- d – number of days
- opt[m][d] – 2d array with max profit for stocks 1..m up to day d.
- best_stock – index of stock with the highest profit
- stock_max – highest profit across all stocks
- min_ind - The index i such that stock m has its **lowest price so far** at day i, for the current day j, where j > i.

  Pseudocode:

```
function DP3(stocks):  // stocks is a 2D array: m stocks, n days

    m = length(stocks)
    d = length(stocks[0])
    create 2D array opt[m][d], initialized to 0
    stock_max = 0
    min_opt = -1
    best_stock = -1

    for stock_index from 0 to m - 1:
        min_ind = 0  // index of minimum price so far

        for day_index from 1 to d - 1:

          //choose max of not selling or selling today
            opt[stock_index][day_index] = MAX(
                opt[stock_index][day_index - 1],
                stocks[stock_index][day_index] -
stocks[stock_index][min_ind]
            )

          // update min_ind if current day has a lower price
            if stocks[stock_index][day_index] <
stocks[stock_index][min_ind]:
                min_ind = day_index

     // check if this stock produced max profit
        if opt[stock_index][d - 1] > stock_max:
            stock_max = opt[stock_index][d - 1]
            best_stock = stock_index
            min_opt = min_ind

   // backtrack to find the sell day
    for i from min_opt + 1 to d - 1:
        if stocks[best_stock][i] - stocks[best_stock][min_opt] ==
stock_max:
            return best_stock, min_opt, i, stock_max  // stock ID,
buy day, sell day, profit
```

```
return 0, 0, 0, 0  // if no profit found
```

Explanation

The algorithm iterates over all stocks, processing each row (stock) sequentially. At each step, it attempts to maximize the transaction amount by either reusing a previously computed transaction or by executing a new transaction, which involves subtracting the minimum value so far from the current value.
The minimum index is reset for each row, and any index i within the current stock row can be a candidate minimum for index j, as long as i < j and stocks[i] = argmin(stocks[i:j-1]).
At the end of each row, the algorithm compares the best local profit for the current stock with the global profit. If the local profit is higher, the global profit, best_stock, and buy_day are updated accordingly.
Once all stocks have been processed, the values for best_stock, buy_day, and profit are already computed. The algorithm then backtracks to identify the corresponding sell_day and finally returns the sequence: best_stock, buy_day, sell_day, and profit.

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

Version 1 – Memory-Intensive Approach

- Stocks – an input 2D array
- s - total number of stocks
- d - total number of days
- opt – a 3D array storing the maximum profit using up to k+1 transactions, considering selling stock m on day n.
- buy – a 2d array that stores the best day(index) to buy stock m for the k-th transaction up to day n.
- sums – a 2D array that stores precomputed profits – sum[m][i][j] - maximum profit from buying stock m on day i and selling on day j
- buyday – the best day index stored in buy[m][n][k]
- k – current transaction index
- n – current day index
- m – current stock index
- diff – the profit difference in backtracking, indicating a new transaction has occurred
- gap – shortest length of interval [buy, sell] in backtracking
- cstock – a candidate stock to reconstruct a transaction during backtracking
- cbuy – a candidate buy day to reconstruct a transaction during backtracking
- transactions – list storing (stock, buy_day, sell_day) tuples

```
function TASK5 (stocks , kval): // stocks is a 2d array, kval is
max allowed transactions
    s =  length(stocks)
    d =  length(stocks[0])
    create 3D array opt[s][d][kval], initialized to 0
    create 3D array sums[s][d][d], initialized to 0
    create 2D array buy[s][kval], initialized to 0
    for index m from 0 to s-1:
        for index i from 0 to d-1: //buy day
            min_val = +INF
            for index j from i+1 to d-1: //sell day
                min_val  = min(min_val, stocks[m][j-1] )
                sums[m][i][j] = stocks[m][j] - min_val

    for index n from 1 to d-1:
        for index m from 0 to s-1:
            for index k from 0 to kval-1: //Index k starts at 0
but represents the 1st transaction

                buyday = buy[m][k]

                //Recurrance relations for different cases
                if(k==0 and m==0):
                     opt[m][n][k]  = sums[m][buyday][n]

                else if(k==0):
                     opt[m][n][k]  = max( opt[m-1][n][k],
sums[m][buyday][n])

                else if(m==0):
                    opt[m][n][k]  = max( opt[m][n-1][k],
                                        opt[m][n-1][k-1] +
sums[m][buyday][n])

                else:
                    opt[m][n][k]  = max( opt[m-1][n][k],
                                        opt[m][n-1][k],
                                        opt[m][n-1][k-1]
+sums[m][buyday][n])

// Update buy day if current price is lower than previous buy day
price
                if( stocks[m][n] <= stocks[m][buyday]) :
                    buy[m][k] = n

                // Update buy day if current price is lower than
previous buy day price and a new transaction has been made
                if(opt[m][n][k] ==  opt[m][n-1][k-1] +
sums[m][buyday][n] ):
                    if k < kval-1:
                        buy[m][k+1] = n

    // Find maximum profit and corresponding transaction count k
    k=-1
    profit = 0
    for index i from 0 to kval-1:
        if (opt[s-1][d-1][i] > profit):
            profit = opt[s-1][d-1][i]
            k = i
    if k==-1:
        return (0, 0, 0)

    sell = length(d)-1
    stock = length(s) -1

  // Find last sell day (where the optimal values are reused)
    transactions = []
    while sell>0:

        // Same profit without using current stock, move up
        if stock >0 and opt[stock][sell][k] == opt[stock -
1][sell][k]:
            stock -=1

        // Same profit as on the previous day, move left
        else if opt[stock][sell][k] == opt[stock][sell-1][k]:
            sell-=1

        else:

// A new transaction has been used: find the stock i and buy day j
// such that selling at day ‘sell’ gives the profit ‘diff’,
// and the interval [j, sell] is the shortest

            diff = opt[stock][sell][k] - opt[stock][sell-1][k-1]
            gap = INF
            cstock = -1
            cbuy = -1
            for index i from stock down to 0:
                for index j from sell down to 0:
                    if diff == stocks[i][sell] - stocks[i][j] and
sell-j < gap:
                        gap= sell-j
                        cstock = i //current stock
                        cbuy = j //current buy

            if cstock==-1 :
                break
            transactions.append((cstock, cbuy, sell))
            sell = cbuy
            k-=1
    return reverse(transactions)
```

Version 2 - The optimized version

Concrete steps will be refined, comments left to explain general logic if there are code inconsistencies.

Variable Definitions:

- Stocks – an input 2D array
- s - total number of stocks
- d - total number of days
- opt – a 3D array storing the maximum profit using up to k+1 transactions, considering selling stock m on day n.
- buy – a 3d array that stores the best day(index) to buy stock m for the k-th transaction up to day n.
- profit – 1d array used for a running best profit for each transaction index k
- buyday – the best day index stored in buy[m][n][k]
- k – current transaction index
- n – current day index
- m – current stock index
- diff – the profit difference in backtracking, indicating a new transaction has occurred
- gap – shortest length of interval [buy, sell] in backtracking
- cstock – a candidate stock to reconstruct a transaction during backtracking
- cbuy – a candidate buy day to reconstruct a transaction during backtracking
- transactions – list storing (stock, buy_day, sell_day) tuples

Pseudocode:

```
Function TASK5 (stocks , kval): // stocks is a 2d array, kval is max allowed transactions
    s =  length(stocks)
    d =  length(stocks[0])
    create 3D array opt[s][d][kval], initialized to 0
    create 3D array buy[s][d][kval], initialized to 0
    create 1D array profit[kval], initialized to 0

    for index n from 1 to d-1:

        for index m from 0 to s-1:

            //Index k starts at 0 but represents the 1st transaction
            for index k from 0 to kval-1:

                buyday = buy[m][n][k]
                profit[k] = max(profit[k], stocks[m][n]  - stocks
[m][buyday])

                //Recurrence relations for different cases
                if(k==0 and m==0):
                     opt[m][n][k] =  max(opt[m][n-1][k], profit[k])

                else if(k==0):
                     opt[m][n][k] = max( opt[m][n-1][k], opt[m-1][n][k],
profit[k] )

                else if(m==0):
                    opt[m][n][k] = max( opt[m][n-1][k],
                                  opt[m][buyday][k-1] + profit[k] )

                else:
                    opt[m][n][k]  = max( opt[m-1][n][k],
                                    opt[m][n-1][k],
                                    opt[m][buyday][k-1] + profit[k])

            // Propagate current buy day to the next day
            buy[m][n+1][k] = buy[m][n][k] //propagate current minimum

                // Update buy day if today's price is lower or equal to
previous buy
                if( stocks[m][n] <= stocks[m][buyday]) :’

                    if( k < kval -1 and n < d-1):

                        buy[m][n+1][k] = n
                        buy[m][n+1][k+1] = n // also update for next
transaction

                // If no new transaction occurred (profit reused), skip
further updates
                if(opt[m][n][k] == opt[m][n-1][k])
                    or opt[m][n][k] == opt[m-1][n][k]
                    or (k>0 and opt[m][n][k] == opt[m][n][k-1]):
                      continue

                // If a new transaction has been made, update the buy day
for the next transaction
                if k < kval-1:
                    buy[m][n+1][k+1] = n

    // Find the maximum profit and the number of transactions used
    k=-1
    profit = 0
    for index i from 0 to kval-1:
        if (opt[s-1][d-1][i] > profit):
            profit = opt[s-1][d-1][i]
            k = i

    if k==-1:
        return (0, 0, 0)
    sell = d- 1
    stock = s-1

  // Find last sell day (where the optimal values are reused)
    transactions = []
    while sell>0:

        // Same profit without using current stock, move up
        if stock >0 and opt[stock][sell][k] == opt[stock -1][sell][k]:
            stock -=1

        // Same profit as on the previous day, move left
        else if opt[stock][sell][k] == opt[stock][sell-1][k]:
            sell-=1

        else:

// A new transaction has been used: find the stock i and buy day j
// such that selling at day 'sell' gives the profit 'diff',
// and the interval [j, sell] is the shortest

            diff = opt[stock][sell][k] - opt[stock][sell-1][k-1]
            gap = INF
            cstock = -1
            cbuy = -1
            for index i from stock down to 0:
                for index j from sell down to 0:
                    if diff == stocks[i][sell] - stocks[i][j] and sell-j <
gap:
                        gap= sell-j
                        cstock = i //current stock
                        cbuy = j //current buy

            if cstock==-1 :
                break
            transactions.append((cstock, cbuy, sell))
            sell = cbuy
            k-=1


    return reverse(transactions)
```

At each step, the algorithm considers three main steps:

- **Exclude the current transaction (adding transaction)**:
  This occurs when including the current transaction does not improve the result. For example, in a strictly decreasing price sequence, it's better to reuse the previous best value from day n-1. In this case, the buy[m][k] (buy day) and cmax (current max) still can reuse the values from the range that is not covered by n-1, unless the current value is the smallest (buy day updated) or the current value is largest (cmax updated).

- **Exclude the current transaction and the current stock:**
  This case occurs when skipping the current stock (row m, opt[m][n-1]) and the current transaction at day n produces a better result. In other words, the result from the previous stock(s) at the same day – opt[m-1][n] is better. This situation arises when earlier stocks (rows 0 to m-1) had transactions with a higher potential profit at day n, compared to what the current stock can offer with and without the newly computed transaction across all stocks up to this point (0 to m rows).
- **Include the result from the current stock + the maximum profit** for the available interval
  In this case, we compute the result as:
  opt[m][buyday][k-1] + profit[k] where profit[k] is the maximum profit that can be obtained across all m stocks up to this point for the available interval [i..j], where i is the last sell day for opt[m][n][k-1]. Since we intend to include a new transaction, we must use the result from at most k-1 transactions up to this point (opt[m][buyday][k-1]) and then add the best profit possible for the k-th transaction (profit[k]).

Because we iterate over the nth day across all stocks (rows) in parallel, it is guaranteed that profit[k] will produce the best possible result from the point of a sale day for the best possible result for j-1, considering all stocks and days up to this point. Since we add profit[k] to opt[m][**buyday**][k-1], and buyday is dynamically updated each time, it is guaranteed that buy day for profit[k] will not overlap with the previous transaction result stored in opt[buyday] for a given k.

Time complexity

The algorithm iterates over all days n for all stocks m, computing a unique optimal profit value for each transaction count k, resulting in a time complexity of O(mnk). Computing profits does add to the overall time complexity. Backtracking is also performed to recover the transaction sequence, with a complexity of (mn^2) ~ n^3.

Therefore, the complexity of the algorithm is O(mnk) ~ O(n^3)

Backtracking

There are three cases:

- **opt[m][n][k] == opt[m][n-1][k]** if the previous result is reused. In this case, we must decrement the sell variable, as no profit was made on the current day, and the profit was reused without making a transaction.
- **opt[m][n][k] == opt[m-1][n][k]** if the previous stock's result is reused. In this case, we must decrement the stock variable, as the current stock produced at most as good a result as the previous stock.
- If none of the above conditions are true, then we must have reached a point where a **transaction occurred**. By the property of our recurrence relation, we know that we must have computed opt[m][n-1][k-1] with the highest profit for the available interval [i..j], where j is greater than or equal to the sell day for opt[m][n-1][k].

However, we do not store any list to recover this information directly. So, we must iterate over all rows of the stocks array and find the minimum range [i..j] that produces the same difference between opt[m][n][k] and opt[m][n-1][k-1].
By finding the smallest range, we can guarantee that there is no better way to compose the current optimal result. And even if there exists a different way to arrive at the same difference between the current optimal and the previous (n-1) optimal, the result obtained is still valid or better.
We exhaustively repeat these steps to recover the entire sequence.
In the end, we reverse the list to preserve the original order of transactions.

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

Assumptions & variable definitions:

- Not allowed to buy any stock until j1 + c + 1
- If an empty array is given, should return tuple of (0,0,0)
- J1 = day stock is bought
- J2 = day stock is sold
- J1 < J2
- C = cooldown period
- I = represents stock index
- M = number of rows/stocks
- N = number of columns/days
- Profit = A[i][j2] – A[i][j1]]

Pseudocode:

```
def bestProfit(A, c):
    initialize m number of stocks
    initialize n number of days

    initialize dp[from 0 to n] with values of 0
    initialize transactions[from 0 to n] to None
    initialize results list empty
    initialize currDay = n – 1

    # Base
    if A is empty or m == 0 or n == 0:
        return empty list

    # get pairs(j1,j2) for each stock index i
    for i from 0 to m-1:
        for j1 from 0 to n-2:
            for j2 from j1+1 to n-1:
                # sell day – buy day
                profit = A[i][j2] - A[i][j1]
                # check if profit is positve
                if profit < 0:
                    #skip profit
                    continue

                # nextTransaction = j1 + c + 1
                prevDay = j1 - c - 1

                if prevDay >= 0:
                    totalProfit = profit + dp[prevDay]
                else:
                    totalProfit = profit
                if totalProfit > dp[j2]:
                    dp[j2] = totalProfit
                   # store transaction
                    transactions[j2] = (i,j1, j2)
                else:
                    dp[j2] = max(dp[j2], dp[j2-1])

    # Backtrack to get best sequence
    while currDay >= 0:
        if transactions[currDay] is not None:
            (i,j1,j2) = transactions[currDay]
            # increment by 1 for base 1 index
            push (i + 1,j1 + 1 ,j2 + 1) to results
            #  move to last allowed transaciton day
            currDay = j1 - c - 1
        else:
            # skip to previous day
            currDay = currDay - 1



    return results
```

This algorithm first checks every profitable pair of days (j1, j2) for each stock index in the matrix A. It uses dp to track the maximum profit that can be achieved on each day while considering the constraint c (cooldown period). Once the dp array is filled, we backtrack from the last day to retrieve the best sequence of transactions that yields the maximum profit.

2.7 Task-7: Dynamic Programming Algorithm for Problem3 O(m·n)

Assumptions & variable definitions:

profitByDay is an array tracking best profit by day

profitByStock tracks best profit so far for a stock

Decision and purchases array hold information on day purchased and sold

Pseudocode:

```
      def bestTrades(A,c):

    m=len(A)
    n=len(A[0])
    profitByDay=[0 for i in range(n)]
    profitByStock=[None for i in range(m)]
    decision=[None]*n
    purchases=[None]*n

    for i in range(n):

        for j in range(m):

            if i>0:

                profitByDay[i]=max(profitByDay[i], profitByDay[i-1])

            if profitByStock[j] != None:
                temp=A[j][i]+profitByStock[j]

                if temp>profitByDay[i]:

                    profitByDay[i]=temp
                    decision[i]=(j,purchases[j][0],i)
                    purchases[j]=None
                    profitByStock[j]=None

            if i-c-1>=0:

                profit = profitByDay[i-c-1]-A[j][i]

                if profitByStock[j] == None or profitByStock[j]<profit:
                    profitByStock[j]=profit
                    purchases[j]=(i,A[j][i])

            elif i-c-1 < 0:
                profit = -A[j][i]
                if profitByStock[j] == None or profitByStock[j]<profit:
                    profitByStock[j]=profit
                    purchases[j]=(i,A[j][i])

    print(decision)
    tuplesToReturn=[]
    i=n-1
    while i>=0:
        print(n)
        print(i,decision[i])
        if decision[i] != None:
            tuplesToReturn.append((decision[i][0],decision[i][1],decision
[i][2]))
            i=decision[i][1]-c-1
        else:
            i=i-1


    return tuplesToReturn
```

The algorithm runs down the rows of stocks first and then across the columns of days resulting in a runtime of O(m*n). It then works similarly to the interval scheduling problem where an asymptotically smaller function backtracks to find best transactions respecting the time limit placed on buy vs selling.

MILESTONE 3

Task 1:

Modifications

No modifications were made to the original pseudocode algorithm.

Assumptions & Limitations

Assumptions:

- The input A is a 2D matrix where A[i][j] will i represents the stock and j represents the day.
- Each stock can be bought and sold once ensuring only one transaction per stock
- Each transaction must follow the condition j1 < j2, where the buy day is before the sell day
- Return maximum profit from a single transaction on a single stock index

Limitations:

- Only can consider one transaction per stock
- Can't combine profits across multiple stocks
- No constraints for number of transactions, and cooldown period since only one transaction is permitted

Description of Implementation (with Proof)

For this function we used a brute force approach to find the most profitable single transaction in the matrix A. We first iterate through each stock i and for each stock we try every possible pair of buy day and sell day (j1, j2) where j2 > j1. For each valid pair, we calculate the profit A[i][j2] – A[i][j1] and keep track of the maximum profit found with the current stock index, buy day, and sell day. By checking all possible buy and sell pairs for every stock, the algorithm ensures that we find the global maximum profit in the matrix A.

Time Complexity

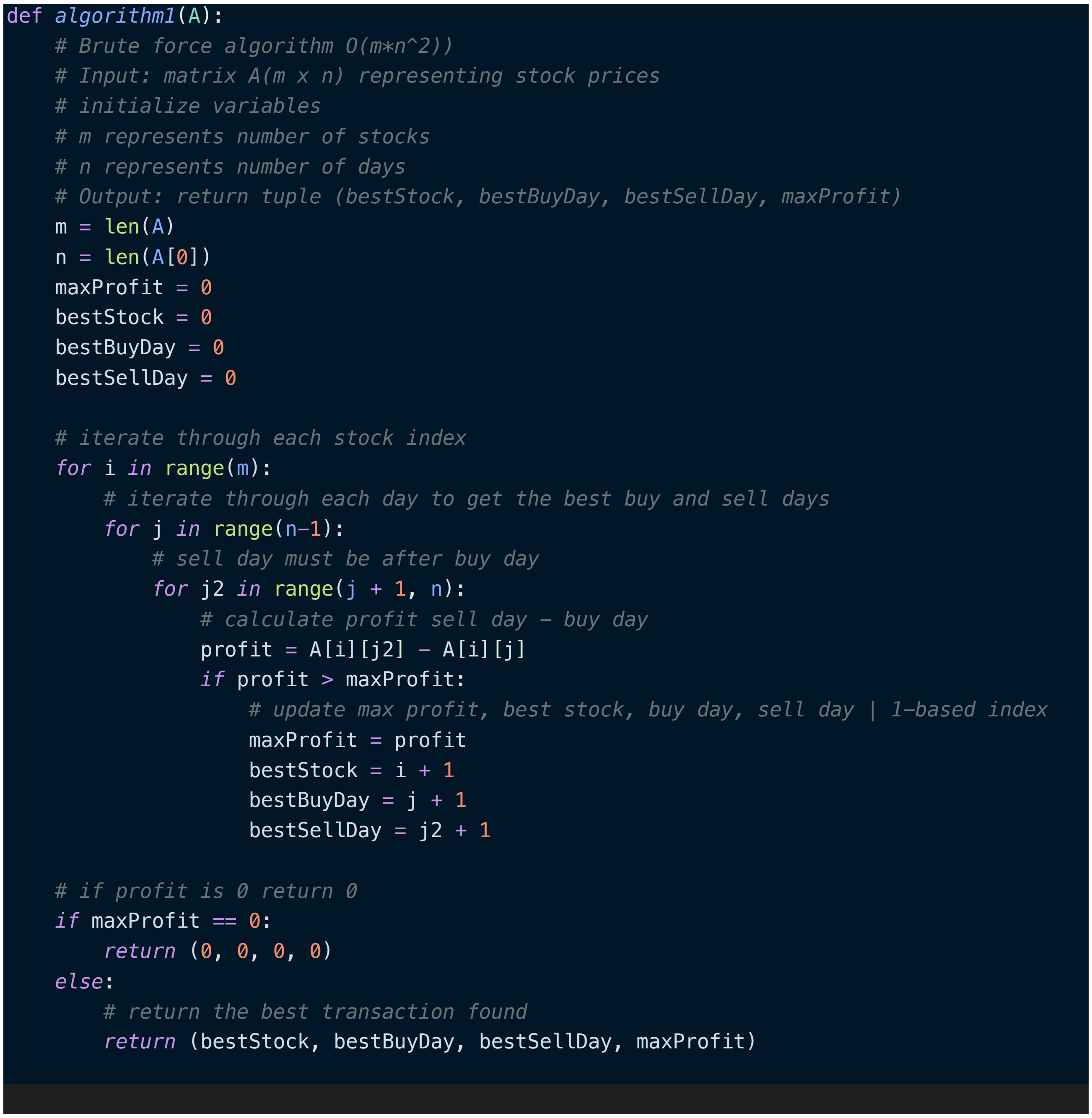
The time complexity for this algorithm is O(m*n^2) where m represents the number of stocks and n represents the number of days. For our brute force approach, we used three nested loops where the outer loop runs m times. The inner two loops check all possible pairs of day (j1, j2) which will run for O(n^2). Therefore, the overall time complexity will be quadratic for each stock in the matrix A running O(m*n^2).

Comparative Analysis with Other Implementations

The brute force algorithm is simpler to implement and ensures that we find the maximum profit by checking each possible buy/sell pair for each stock. It does however run in O(m*n^2) time, which will become slower as the number of days increases. When comparing this algorithm to the greedy approach in task 2, we find that the greedy algorithm is more efficient running O(m*n). The DP solution also runs in this time complexity making both the greedy algorithm and DP algorithm more efficient. These algorithms would be better than the brute-force approach for larger input sizes. The brute-force algorithm like the greedy algorithm do use less memory than the DP making them both more space efficient.

Trade-off Discussion

The brute force approach is much easier to implement and understand compared to possible algorithms. It checks all valid transactions and finds the best one which would be better suited for smaller datasets. Since it checks every possible pair extensively, the computation time is quadratic O(m*n^2) making it less efficient when the input size for n is increased. Using a different approach like the greedy or DP method would be more complex in designing it but would be more efficient as they both run linear time O(m*n). These algorithms would perform and be better for larger input sizes than the brute-force method. The brute-force algorithm like the greedy algorithm both use less memory than the DP method in this regard.



```python
def algorithm1(A):
    # Brute force algorithm O(m*n^2))
    # Input: matrix A(m x n) representing stock prices
    # initialize variables
    # m represents number of stocks
    # n represents number of days
    # Output: return tuple (bestStock, bestBuyDay, bestSellDay, maxProfit)
    m = len(A)
    n = len(A[0])
    maxProfit = 0
    bestStock = 0
    bestBuyDay = 0
    bestSellDay = 0

    # iterate through each stock index
    for i in range(m):
        # iterate through each day to get the best buy and sell days
        for j in range(n-1):
            # sell day must be after buy day
            for j2 in range(j + 1, n):
                # calculate profit sell day - buy day
                profit = A[i][j2] - A[i][j]
                if profit > maxProfit:
                    # update max profit, best stock, buy day, sell day | 1-based index
                    maxProfit = profit
                    bestStock = i + 1
                    bestBuyDay = j + 1
                    bestSellDay = j2 + 1

    # if profit is 0 return 0
    if maxProfit == 0:
        return (0, 0, 0, 0)
    else:
        # return the best transaction found
        return (bestStock, bestBuyDay, bestSellDay, maxProfit)
```

Task 2:

Modifications

There were no modifications made to the original pseudocode implementation.

Assumptions & Limitations

Assumptions:

- The input A is a 2D matrix where A[i][j] will i represents the stock and j represents the day.
- Each stock can be bought and sold once
- Buy day must be before sell day (j1 ,< j2)
- Return maximum profit form a single transaction on a single stock

Limitations:

- Only one transaction per stock is allowed
- Can't combine profits across multiple stocks
- No constraints for number of transactions, and cooldown period

Description of Implementation (with Proof)

This algorithm uses a greedy approach to find the maximum profit for each stock transaction in linear time. We first iterate through each stock and for each stock we initialize the minPrice to the price of the first day and the minDay to day 0. Then we iterate through the remaining days in the stock and check if the current price minus the minPrice yields a better profit than the maxProfit. If it does, we update the maxProfit to this value and update the best stock day, best buy day, and best sell day as well. If the current price is less than the minPrice, we update the minPrice and minDay.

This ensures that for each stock the algorithm buys at the lowest seen price, and checks whether selling at the current day would result in the highest potential profit. If there a no profitable transaction, our algorithm returns a tuple (0,0,0,0), otherwise we return the stock index, buy day, sell day, and the max profit. Since the algorithm checks all days for each stock, and tracks the lowest current price, it ensures that we return the best transaction for maximum profit.

Time Complexity

The time complexity for this algorithm is O(m*n) where m represents the number of stocks and n represents the number of days. The outer loop iterates for each stock m times, and the inner loops runs n times for each day. This is more efficient than the brute-force approach which has a time complexity of O(m*n^2).

Comparative Analysis with Other Implementations

The greedy algorithm results in a more efficient time complexity than the brute force method. The brute force method runs at O(m*n^2) since it tries all possible pairs (buy day, sell day), whereas the greedy algorithm tracks the lowest price day and iterates through each stock to find the best transaction. It matches the performance for the DP solution in task 3, but it uses less memory and is easier to implement for this problem.

Trade-off Discussion

The greedy algorithm is highly efficient and would be better suitable for larger input sizes due to its performance. The greedy algorithm is more complex than the brute force approach since when need to track the lowest price day to maximize our profit and to return the best transaction. It does however reduce the time complexity, making it a better option for performance. Also, unlike the DP approach which runs at the same time complexity, the greedy algorithm uses less memory making it the better overall choice.

```python
def algorithm2(A): #MaxProfitGreedy
#Input: matrix A(m × n) representing stock prices
# m represents number of stocks
# n represents number of days
#Output: tuple (bestStock, bestBuyDay, bestSellDay, maxProfit) representing the best
stock and days to buy/sell and max profit
    m = len(A)
    n = len(A[0])
    maxProfit = 0
    bestStock = 0
    bestBuyDay = 0
    bestSellDay = 0

    # iterate through each stock index
    for i in range(m):
        # initialize minPrice and minDay for this stock
        minPrice = A[i][0]
        minDay = 0

        # iterate through each day for current stock
        for j in range(1, n):
            # if selling gives a better profit than maxProfit, update best transaction
            if A[i][j] - minPrice > maxProfit:
                maxProfit = A[i][j] - minPrice
                # add 1 for 1-based index
                bestStock = i + 1
                bestBuyDay = minDay + 1
                bestSellDay = j + 1

            # if current price is lower than minPrice we update
            # minPrice and minDay to current day and price
            if A[i][j] < minPrice:
                minPrice = A[i][j]
                minDay = j

    # if profit is 0 then there is no profitable transaction made
    if maxProfit == 0:
        return (0, 0, 0, 0)
    else:
        # returns best transaction found
        return (bestStock, bestBuyDay, bestSellDay, maxProfit)
```

Task 3

Modifications

This approach has a slight modification in the construction of the `opt` array. Unlike the previous implementation, which combined greedy and dynamic strategies, the current implementation relies solely on a dynamic programming approach.

Description of Implementation

We draw the following assumptions:

- We initialize and must update the minimum price (buy day) for each stock separately, as the minimum for one stock cannot be reused for another.
- The minimum price day or buy day (d_b), must always be less than the current sell day (n), just like in a greedy approach.
- If we reduce our problem to a single stock, then at each step we can either sell or not sell. If we sell, we must know the day we bought the stock. If we move linearly from left to right (from day 0 to day n), we can update the buy day on the fly. If we don't sell today, we might have sold yesterday and obtained some profit. Regardless of whether we made a profit yesterday or not, propagating the value forward is always valid:
    - Worst case, we propagate zero if no transaction was possible.
    - Best case, we propagate a better transaction than what would have been obtained by selling on day n.

  This buy day (d_b) should always be at least one day before the current day, so that the constraint d_b, < n always holds. If we move forward in constructing our dynamic table and only update the buy day at the end of each n-iteration, and only if the current price is less than or equal to the price at the previous buy day, then d_b < n is guaranteed.
- Once multiple stocks are introduced, we must also consider the possibility that a better transaction occurred in an earlier stock by this day. In that case, we must propagate that earlier transaction, potentially choosing it over both the current transaction and prior transactions from the current stock.
- For correctness, the buy day must be reinitialized for each stock.

At each step, the following three options compete to define a new value of opt[m][n]:

- $opt[m][n-1]$

$the\ best\ result\ from\ not\ selling\ today\ (carry\ forward\ the\ previous\ day's\ value)$.

- $stocks[m][n]$ – $stocks[m][buyday]$ – $profit\ from\ a\ valid\ sell\ today$.
- $opt[m-1][n]$ – $the\ best\ result\ from\ a\ previous\ stock\ at\ the\ same\ day$.

These three cases ensure an optimal substructure: either we compute a new best result from selling today, or we propagate a previously optimal result from an earlier day or an earlier stock.
Even if the true optimal transaction occurred at some earlier stock $m-i$ and day $n-j$, the above recurrence guarantees that the result will propagate forward correctly to opt$[m][n]$.

The backtracking is based on the following logic:

- If the best transaction occurred at some stock $m-i$ and day $n-j$, its value would propagate to $opt[m][n]$. Since the opt array is constructed relative to the $sellday$, in order to recover the $sellday$, we must find the indeces $m-i$ and $n-j$. Both $m-i$ and $n-j$ are the first point moving backwards where opt changes along both m and n directions. Thus, by scanning backwards, we can recover the correct sell day.
- Furthermore, because values from $opt[m–i][n–j]$ propagate forward into $opt[m][n]$, the final index $opt[m][n]$ $(or\ opt[m-1][n-1]$ in zero-based notation) will always contain the maximum profit achievable across all stocks and all days.
  Once we recover the $sellday$ and we know the maximum $profit$, we can iterate backward over that stock's prices to identify the buy day. This is done by checking the following equality:
  $stocks[m][d\_s]$ – $stocks[m][i]\ ==\ profit, where\ i < d\_s$
  or
  $stocks[m][d\_s]$ – $stocks[m][i]\ ==\ opt[m][n], where\ i < d\_s$

Time Complexity

The algorithm iterates over all stocks m, and for each stock, it iterates over all its days n. The operations inside the innermost loop to construct the opt array involve constant-time comparisons and accesses. Therefore, the construction of the opt array has a time complexity of O(mn).
Backtracking is also performed with the same complexity. We first iterate over every stock and then, for each stock, iterate over all of its days.
Thus, the total time complexity is O(mn) ~ O(n^2).

Comparative Analysis with Other Implementations

This algorithm is an optimized dynamic programming implementation. Compared to the naïve approach, which has a time complexity of O(n^3), this implementation reduces it to O(n^2), resulting in a significant runtime improvement for large inputs. While it performs fewer operations, it requires slightly more space.
However, when compared to the greedy algorithm, the greedy method outperforms this dynamic approach. Although both have time complexity of ~ O(n^2) the greedy algorithm uses less memory

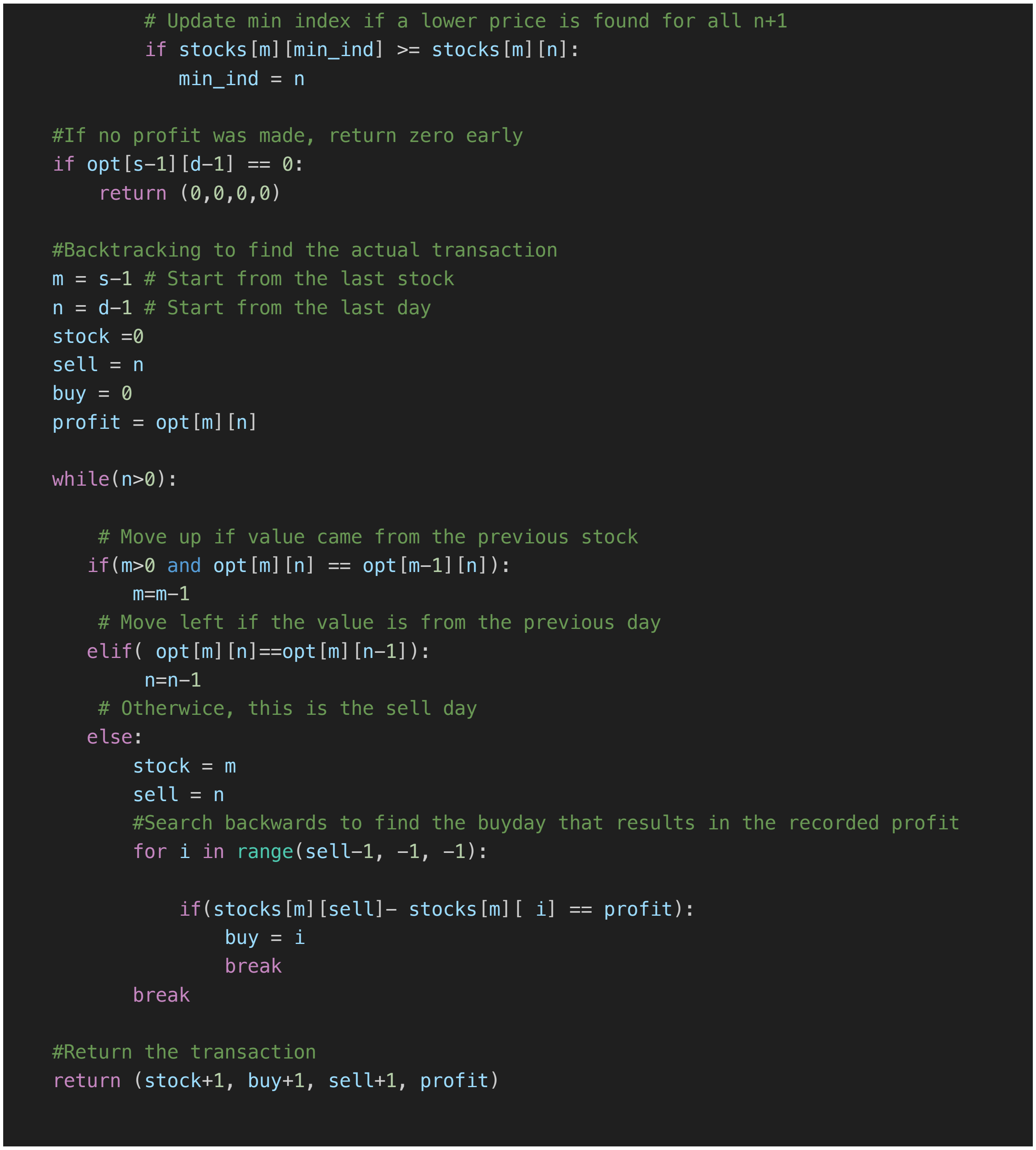
by tracking the minimum day on the fly and updating the best result after processing each stock. This eliminates the need to allocate an mn opt array.
Therefore, in terms of practical efficiency, the greedy approach is generally the best choice for both small and large inputs.

Trade-off Discussion

There is a significant space-runtime tradeoff involved in this algorithm. As mentioned earlier in the comparative analysis, the greedy version is the most optimal. The dynamic solution is too complex for this type of problem and requires more memory than a greedy approach would. While not critical on modern machines, for large inputs, the size of the opt array can reach millions given task constraints, whereas the greedy method avoids creating such a large array altogether. This reduces both memory usage and runtime needed to execute the program.

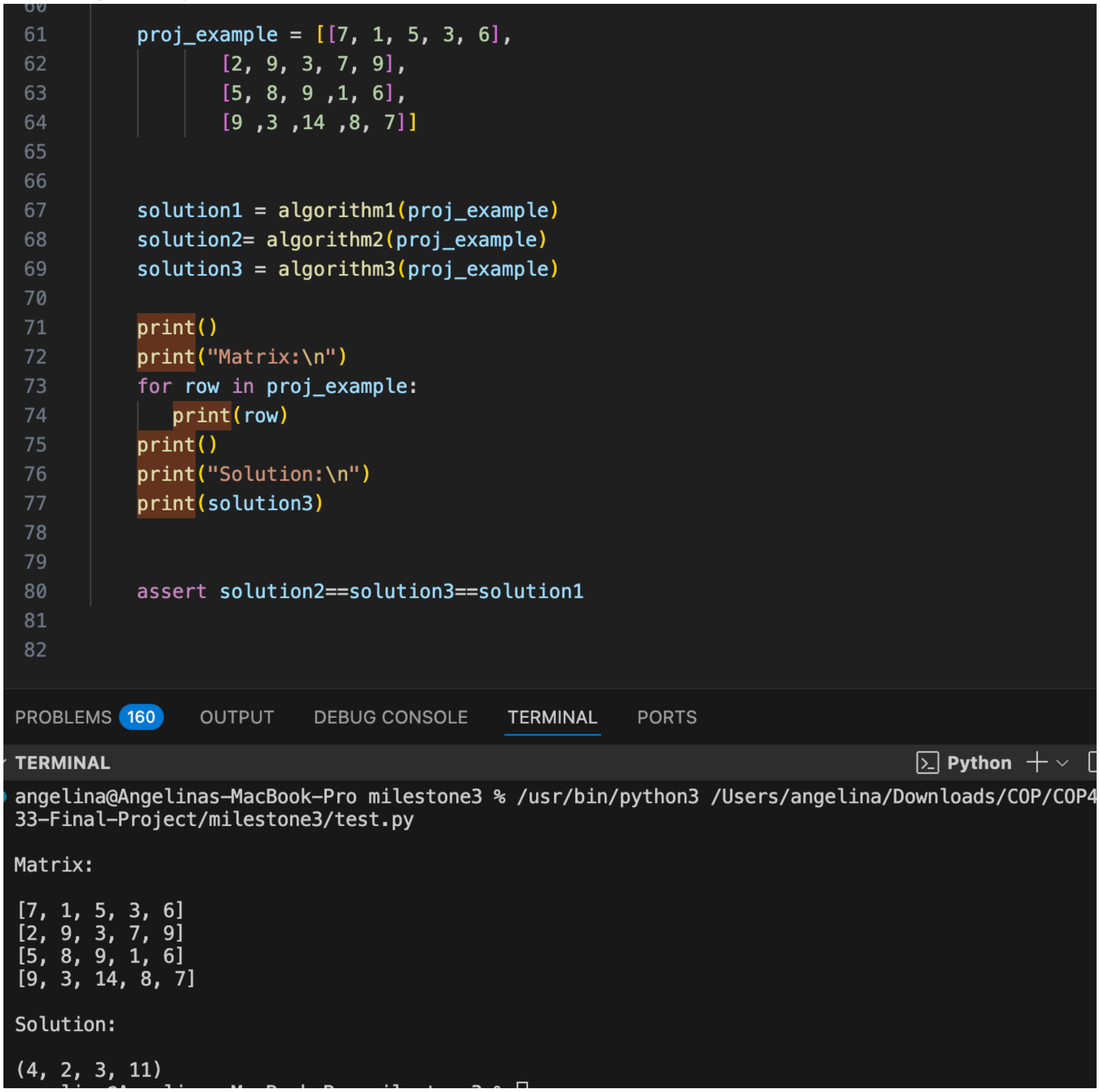
Code

```
def algorithm3 (stocks):

    # 2d DP array to store optimal profits where each index(day) represents a potential
sell day for the current stock
    opt = [ [0 for _ in range(len(stocks[0]))] for _ in range(len(stocks)) ]
    s = len(stocks)
    d = len(stocks[0])

    #Build a DP array opt
    for m in range(s):
       # Initialize index of the buy day with the minimum price so far for stock m
       min_ind = 0

      #Iterate over each day, starting from 1, since no profit can be made on the first
day
       for n in range(1, d):
            # Calculate max profit using:
            # - Previous day profit on the same stock
            # - Profit up to the same day from any of the previous stocks
            # - Profit form selling today using min_ind as buyday
            if m>0:
                opt[m][n] = max(opt[m][n-1], opt[m-1][n], stocks[m][n] -
stocks[m][min_ind])
            else:
                 opt[m][n] = max(opt[m][n-1], stocks[m][n] - stocks[m][min_ind])
```



```
        # Update min index if a lower price is found for all n+1
        if stocks[m][min_ind] >= stocks[m][n]:
           min_ind = n

#If no profit was made, return zero early
if opt[s-1][d-1] == 0:
    return (0,0,0,0)

#Backtracking to find the actual transaction
m = s-1 # Start from the last stock
n = d-1 # Start from the last day
stock =0
sell = n
buy = 0
profit = opt[m][n]

while(n>0):

    # Move up if value came from the previous stock
   if(m>0 and opt[m][n] == opt[m-1][n]):
       m=m-1
    # Move left if the value is from the previous day
   elif( opt[m][n]==opt[m][n-1]):
        n=n-1
    # Otherwice, this is the sell day
   else:
       stock = m
       sell = n
       #Search backwards to find the buyday that results in the recorded profit
       for i in range(sell-1, -1, -1):

           if(stocks[m][sell]- stocks[m][ i] == profit):
               buy = i
               break
       break

#Return the transaction
return (stock+1, buy+1, sell+1, profit)
```

Testing using an example from the project file.



```
    proj_example = [[7, 1, 5, 3, 6],
            [2, 9, 3, 7, 9],
            [5, 8, 9 ,1, 6],
            [9 ,3 ,14 ,8, 7]]


    solution1 = algorithm1(proj_example)
    solution2= algorithm2(proj_example)
    solution3 = algorithm3(proj_example)

    print()
    print("Matrix:\n")
    for row in proj_example:
        print(row)
    print()
    print("Solution:\n")
    print(solution3)


    assert solution2==solution3==solution1
```

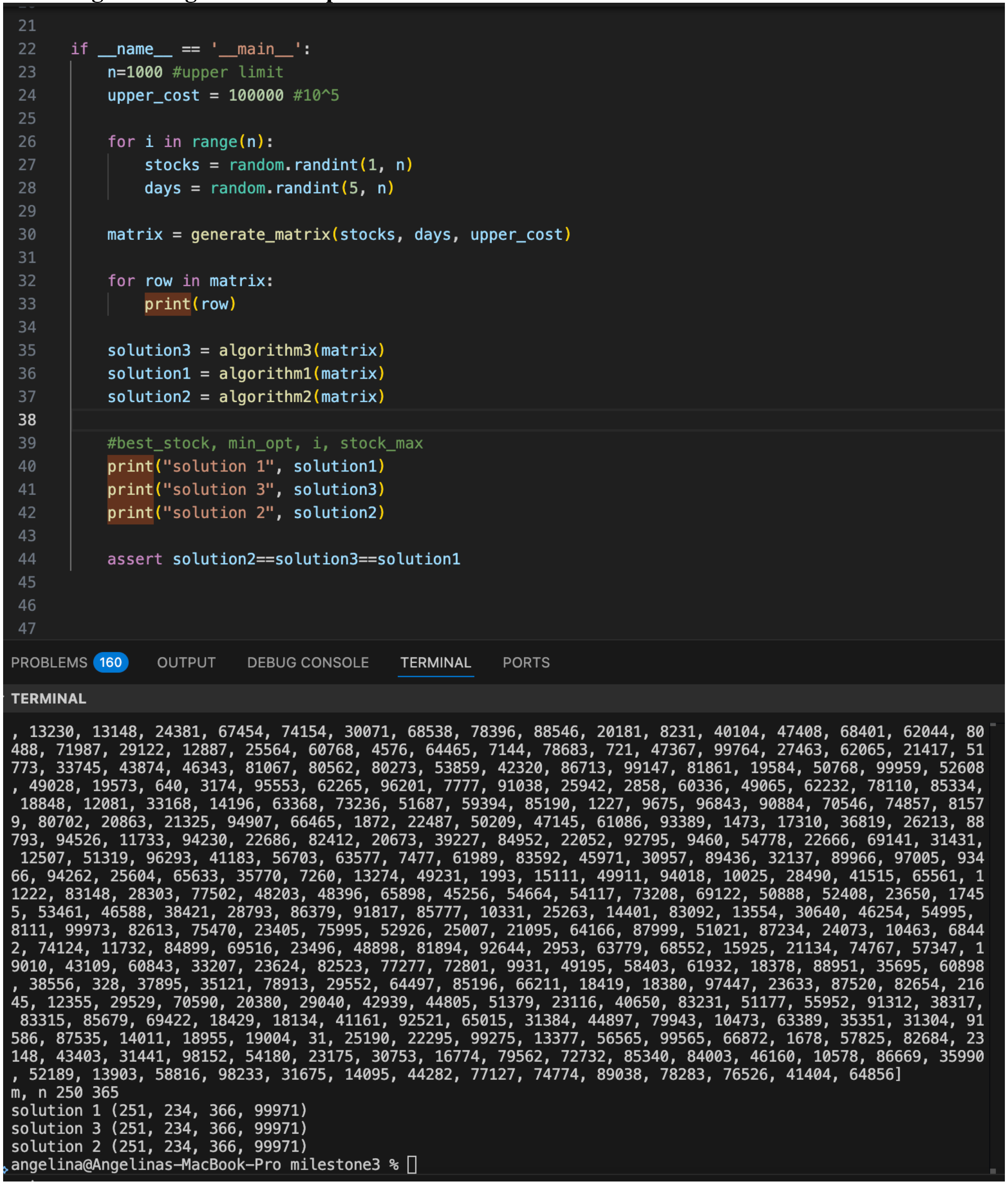
PROBLEMS 160 OUTPUT DEBUG CONSOLE TERMINAL PORTS

TERMINAL Python

```
angelina@Angelinas-MacBook-Pro milestone3 % /usr/bin/python3 /Users/angelina/Downloads/COP/COP4
33-Final-Project/milestone3/test.py

Matrix:

[7, 1, 5, 3, 6]
[2, 9, 3, 7, 9]
[5, 8, 9, 1, 6]
[9, 3, 14, 8, 7]

Solution:

(4, 2, 3, 11)
```

Verify that all three algorithms perform correctly and produce identical results

when given large random input



```python

if __name__ == '__main__':
    n=1000 #upper limit
    upper_cost = 100000 #10^5

    for i in range(n):
        stocks = random.randint(1, n)
        days = random.randint(5, n)

    matrix = generate_matrix(stocks, days, upper_cost)

    for row in matrix:
        print(row)

    solution3 = algorithm3(matrix)
    solution1 = algorithm1(matrix)
    solution2 = algorithm2(matrix)

    #best_stock, min_opt, i, stock_max
    print("solution 1", solution1)
    print("solution 3", solution3)
    print("solution 2", solution2)

    assert solution2==solution3==solution1



```

```
, 13230, 13148, 24381, 67454, 74154, 30071, 68538, 78396, 88546, 20181, 8231, 40104, 47408, 68401, 62044, 80
488, 71987, 29122, 12887, 25564, 60768, 4576, 64465, 7144, 78683, 721, 47367, 99764, 27463, 62065, 21417, 51
773, 33745, 43874, 46343, 81067, 80562, 80273, 53859, 42320, 86713, 99147, 81861, 19584, 50768, 99959, 52608
, 49028, 19573, 640, 3174, 95553, 62265, 96201, 7777, 91038, 25942, 2858, 60336, 49065, 62232, 78110, 85334,
 18848, 12081, 33168, 14196, 63368, 73236, 51687, 59394, 85190, 1227, 9675, 96843, 90884, 70546, 74857, 8157
9, 80702, 20863, 21325, 94907, 66465, 1872, 22487, 50209, 47145, 61086, 93389, 1473, 17310, 36819, 26213, 88
793, 94526, 11733, 94230, 22686, 82412, 20673, 39227, 84952, 22052, 92795, 9460, 54778, 22666, 69141, 31431,
 12507, 51319, 96293, 41183, 56703, 63577, 7477, 61989, 83592, 45971, 30957, 89436, 32137, 89966, 97005, 934
66, 94262, 25604, 65633, 35770, 7260, 13274, 49231, 1993, 15111, 49911, 94018, 10025, 28490, 41515, 65561, 1
1222, 83148, 28303, 77502, 48203, 48396, 65898, 45256, 54664, 54117, 73208, 69122, 50888, 52408, 23650, 1745
5, 53461, 46588, 38421, 28793, 86379, 91817, 85777, 10331, 25263, 14401, 83092, 13554, 30640, 46254, 54995,
8111, 99973, 82613, 75470, 23405, 75995, 52926, 25007, 21095, 64166, 87999, 51021, 87234, 24073, 10463, 6844
2, 74124, 11732, 84899, 69516, 23496, 48898, 81894, 92644, 2953, 63779, 68552, 15925, 21134, 74767, 57347, 1
9010, 43109, 60843, 33207, 23624, 82523, 77277, 72801, 9931, 49195, 58403, 61932, 18378, 88951, 35695, 60898
, 38556, 328, 37895, 35121, 78913, 29552, 64497, 85196, 66211, 18419, 18380, 97447, 23633, 87520, 82654, 216
45, 12355, 29529, 70590, 20380, 29040, 42939, 44805, 51379, 23116, 40650, 83231, 51177, 55952, 91312, 38317,
 83315, 85679, 69422, 18429, 18134, 41161, 92521, 65015, 31384, 44897, 79943, 10473, 63389, 35351, 31304, 91
586, 87535, 14011, 18955, 19004, 31, 25190, 22295, 99275, 13377, 56565, 99565, 66872, 1678, 57825, 82684, 23
148, 43403, 31441, 98152, 54180, 23175, 30753, 16774, 79562, 72732, 85340, 84003, 46160, 10578, 86669, 35990
, 52189, 13903, 58816, 98233, 31675, 14095, 44282, 77127, 74774, 89038, 78283, 76526, 41404, 64856]
m, n 250 365
solution 1 (251, 234, 366, 99971)
solution 3 (251, 234, 366, 99971)
solution 2 (251, 234, 366, 99971)
angelina@Angelinas-MacBook-Pro milestone3 %
```

Task 5

Modifications

- **The gap variable was removed** because, with each iteration, the search range only widens. Therefore, once a valid matching interval is found, continuing the search would be redundant and potentially incorrect, as any subsequent interval would be wider and possibly not optimal anymore.
- In edge cases where the buy day did not result in the best $k - th$ transaction sum, we realized that the **interval must be counted from the $\boldsymbol{sellday}$** (where $k - 1th$ transaction ended) rather than the buy day for the current transaction, stock, and sellday. This is because the buy day can be updated multiple times within the same row if the minimum stock price continues to decrease. If we rely solely on the buy day, we might exclude more profitable intervals from earlier rows that still fall within valid bounds (greater or equal to $sellday\ for\ k - 1$).
- **The forward propagation of the minimum $\boldsymbol{buyday}$ for transaction count $\boldsymbol{k}$ was refined** after observing that updating it immediately leads to incorrect behavior for k+1. To address this, we must defer updating it and update only after completing all iterations over K for the currently simulated sell day n. The minimum should be propagated to all n+1 columns(days) for future transactions $k + i + 1$ $(where\ i >= 0\ and\ k + i + 1 <= K)$, but only when performing a new kth transaction and selling on the current day n results in a global improvement. Therefore, each $k + 1 - th$ transaction must begin no earlier than the k-th transaction's sell day.
- **The backtracking logic was revised** to align with an updated forward propagation, allowing each $opt[m][n][k]$ value to potentially be computed from the largest valid interval starting at the sell day of the $k - 1th$ transaction, across any row $i < m$ up to the current point.

Assumptions & Limitations

There are no memory constraints; this implementation assumes adequate memory availability on modern machines.

Description of Implementation

Let:

$d_s = sell\ day\ from\ k - 1$
$d_b = buy\ day\ for\ k$
$n = currently\ simulated\ sell\ day\ for\ stock\ m < M$
$M = total\ number\ of\ stocks$
$K = total\ numbr\ of\ allowed\ transactions$

$N \ = total\, number\, of\, days$
Type equation here.

We define the size of the opt array based on this observation:

We notice that at each step we are allowed to perform at most k transactions. For each stock and each day, we can perform up to k transactions. Since the optimal number of transactions is not predictable in advance, we must simulate all possible transaction counts up to k.

Recurrence Relation

At each $opt[m][n][k]\ where\ m < M, n < N, k < K,$ at most 4 candidates compete to form the optimal profit:

- $opt[m][n-1][k]$ – continue with the same number of transactions, selling on some day before
- $opt[m][b\_d][k-1]\ +\ sums[m][buyday][n]$ – Best profit for (k-1) transactions ending on a valid buy day plus profit from the current transaction (accounts for a single stock)
- $opt[m][s\_d][k-1]\ + sums[m][sellday][n]$ $-$ Best profit from (k-1) transactions plus the best interval from any earlier sell day from any stock up to this point.
- $opt[m][n][k-1]$ – optimal value can be achieved with fewer transactions.

A decision at step [m][n][k] incorporates and builds upon all previous decisions made over the range of days $[0, n)$ $(where\ n\ <\ N)$ and stocks $[0, m)$ $(where\ m\ <\ M)$.

Key Cases:

The derivation of $\boldsymbol{opt[m][n-1][k]\ and\ opt[m][n][k-1]}$ is relatively straightforward These case are similar to handling multiple stocks with a maximum of one transaction are handles, with an addition of a third dimensions to account for the number of allowed transactions.

$\boldsymbol{opt[m][buyday][k-1]\ +\ sums[m][buyday][n]}$

This case considers only the current stock row. Since we are constrained to at most k transactions, we must use the optimal profit from k-1 transactions and add the best profit from a valid buy-sell interval that follows $k-1th$ upper bound.
We assume that the sell day d_s from the k$-1th$ transaction produced the best result for the $[m][n]$ range.
Thus, for the kth transaction, the buy day must occur on or after d_s. This is enforced through $buyday$ and $sellday$ arrays, updated during the construction of the opt under specific conditions, discussed later.
Assuming this logic is completely implemented, and the minimum buy day is correctly tracked, let d_b be the minimum buy day for the interval $d_s, n-1]$ (where d_s is $k-1th\ sellday$ and upper bound and kth lower bound). In this case, the optimal additional profit for the k-th transaction

would be the maximum difference in the range $[d_b, n]$, because $d_b >= d_s$, and d_b has the smallest price in the range [d_s, n-1].

The maximum profit is computed as follows:

$\boldsymbol{stocks[m][n]}$ – $\boldsymbol{stocks[m][d\_b]}$.

We cannot use the sums array here because it includes data from all stocks (0 to m), not just the current one. To get the profit for the current stock only, we must compute it directly using $stocks[m][n] - stocks[m][buyday]$.
Let's denote the difference as profit. Then the total profit for k transactions becomes:

$\boldsymbol{opt[m][d_b][k-1] + profit}$

Here, $opt[m][d_b][k-1]$ is guaranteed to be the best result from k-1 transactions ending on d_b,, which was computed using the same recursive logic.

$opt[m][sellday][k-1] + sums[m][sellday][n]$

This is the most complex case that build upon the logic of the previous case but adapts it for the multiple stocks.
In the single-row case, the difference $stocks[m][n] - stocks[m][\ d_b]$ (from the current day and the minimum buy day) is guaranteed to be greater than $stocks[m][n]$ – $stocks[m][\ d_s]$ or any difference involving the previous sell day. However, with multiple stocks, this assumption no longer holds.
Now we must consider the maximum possible profit across all [0, m] stocks in the range [d_s, n], and by using current row's $buyday\ d_s$ as lower bound for our computation, the algorithm may miss better opportunities from the upper stocks. Therefore, we need to simulate all valid intervals [d_s, n] for every stock in [0, m].
To support hits, we separately track $sellday(d_s\ )$ for $k-1$ and propagate it to the k-th transaction. Using this, we can simulate selling on day n and buying on $k-1st\ d_s$ for m-th row across all stocks. The design of the sums array guarantees that for any interval [d_s, n], the maximum profit across all stocks up to m adheres to the following property:

$sums[m][\ d_s][n] >= sums[m-1][\ d_s][n]$

The combined profit is calculated as follows:
$opt[m][sellday][k-1] + sums[m][sellday][n]$

Where:

- $opt[m][\ d_s][k-1]$ is the best profit using $k-1$ transactions,

- $sums[m][\ d_s][n]$ is the best possible profit from an additional transaction across any stock in $[0, m]$ over the interval $[d_s, n]$.

Minimum buyday tracking and forward propagation

Now, we discuss the tracking and forward propagation of the minimum buy day (d_b) which is crucial for correctly computing the opt array. When we find a result that improves upon all previous results up to this point, it guarantees that no better combination exists to achieve this result. Therefore, we must mark this point as the start of a new transaction. Consequently, all subsequent transactions must begin from this updated minimum buy day, so we update the minimum buy day accordingly.

However, if we update it immediately within the innermost loop, we cannot correctly simulate transactions at $k + 1$ for the current day i $(the\ current\ sell\ day)$. The update should only affect day $n + 1$, not day n. To handle this, we track the first occurrence of an improvement and defer updating the minimum buy day for all transactions $k + i < K$, starting from the transaction immediately after the one where the improvement occurred.

We update only the next transaction $k + 1$ because an improvement at transaction k does not affect the current simulation at day i and transaction K. On the next day i+1, when simulating k transactions, the minimum buy day will be updated if a smaller buy day is encountered. Crucially, we avoid updating the buy day for the current kth transaction on the same day as the improvement, since this update concerns only all subsequent transactions starting from this point.

Sellday tracking

We also must track $sellday\ for\ k - 1$ and update it for each k. $Sellday$ is not directly useful within the actual transaction at which it occurs, but it plays an important role in defining boundaries between transactions. When a new transaction that results in a global improvement occurs at $k - 1$, we know that the current n day becomes a $buyday\ d_b$ for transaction k. So, we propagate this way as the new buy day and the sell day for the kth transaction. It is important to track $sellday\ d_s$ and $buyday\ d_b$ separately because d_b can be repeatedly updated if a smaller price is found for transaction k, whereas d_s needs to be preserved to maintain the original sell boundary of the $k - 1th$ transaction. Tracking d_s allows us to simulate a new transaction within the largest valid region, consistent with the task's requirement that the kth transaction can only begin on or after the sell day of the $k - 1th$ transaction.

Backtracking logic

As previously discussed, at each $opt[m][n][k]$ position, the result is determined by one of the following four cases:

- $opt[m][n-1][k]$,
- $opt[m][d\_b][k-1] + sums[m][d\_b][n]$
- $opt[m][d\_s][k-1] + sums[m][d\_s][n]$,
- $opt[m][n][k-1]$

Since $opt[m][n][k]$ always reflects selling on day n, if the value does not match the value from any of the prior options ($n-1, k-1, or\ m-1$), it means that a new sell occurred on the current day n that improved the global result.

In this case, we have two following subcases to handle:

- First transaction ($k = 0$ in zero based indexing)
- $k > 0$

First transaction (k = 0 in zero based indexing)

If we are at the first transaction ($k = 0$), then $opt[m][n][0]$ represents the best profit obtainable using only a single transaction. Unlike the case for k > 0, this is not an added profit (it can be treated as starting from zero). Therefore, to recover the corresponding buy day d_b , we use the following equality:

$$stocks[i][n] - stocks[i][d\_b] == opt[m][n][0], where\ i <= m$$

We do not know which stock $i < m$ produces this maximum profit, so we must iterate over all stocks in $[0, m]$. Still, because we know that opt array was constructed relative to the current row m, we perform backtracking using opt[m] rather than opt[i].

Second case: k > 0

If none of the other conditions hold, it indicates that a new sale must have occurred on the current day n. Since we are at $k > 0$ transactions, we know that a new profit was added to the previous optimal value opt[m][d_s][k − 1]. To recover the corresponding buy day d_b, we search for an index satisfying the following equality:

$$opt[m][n][k] - opt[m][d_b][k-1] == stocks[i][n] - stocks[i][d_b,],$$ where i < m

This condition ensures that the profit added during the kth transaction corresponds exactly to the price difference between a buy on day d_b, and a sale on day n for some stock i.

However, just like in the previous case, while the added profit could be from any stock i, we construct the optimal solution relative to the $m-th$ stock. Therefore, we must use $opt[m]$ rather than $opt[i]$ to maintain consistency with how the opt array was built.

Time Complexity

The algorithm runs as follows:

Construction of the sums array involves iterating over all stocks and all possible combinations of buy and sell days to precompute the maximum profit over intervals. Time complexity is O(mn^2).

Next, we construct the opt array to keep track of maximum profits. For each transaction count k, for everyday n, and for every stock m, the algorithm updates the optimal profit. All operations inside the loop are constant time. Time complexity is $O(mnk)$.

The backtracking part reconstructs the transaction sequence. The outer loop decreases either the stock index, day, or transaction count based on current opt array state. Under certain conditions, it enters another loop that may iterate over all valid sell days and stock-day combinations within a constrained region. While we do not iterate over the full kmn space again, the nested loops over possible sell days and stock-day pairs lead to the time complexity $of\ O(sellday * m * k)$.

The overall time complexity is O ($\boldsymbol{n^3}$)

Comparative Analysis with Other Implementations

This implementation uses an optimized dynamic approach, reducing the time complexity compared to a naïve implementation with an exponential time complexity. While the algorithm runs in polynomial running, it relies on many helper arrays, one which significantly impacts memory usage. The $sums[m][n][n]$ array can potentially grow to a size of 10^{12} entries, leading to a very high storage requirement. This makes it the most memory-intensive part of the program and a target for optimization.

Trade-off Discussion

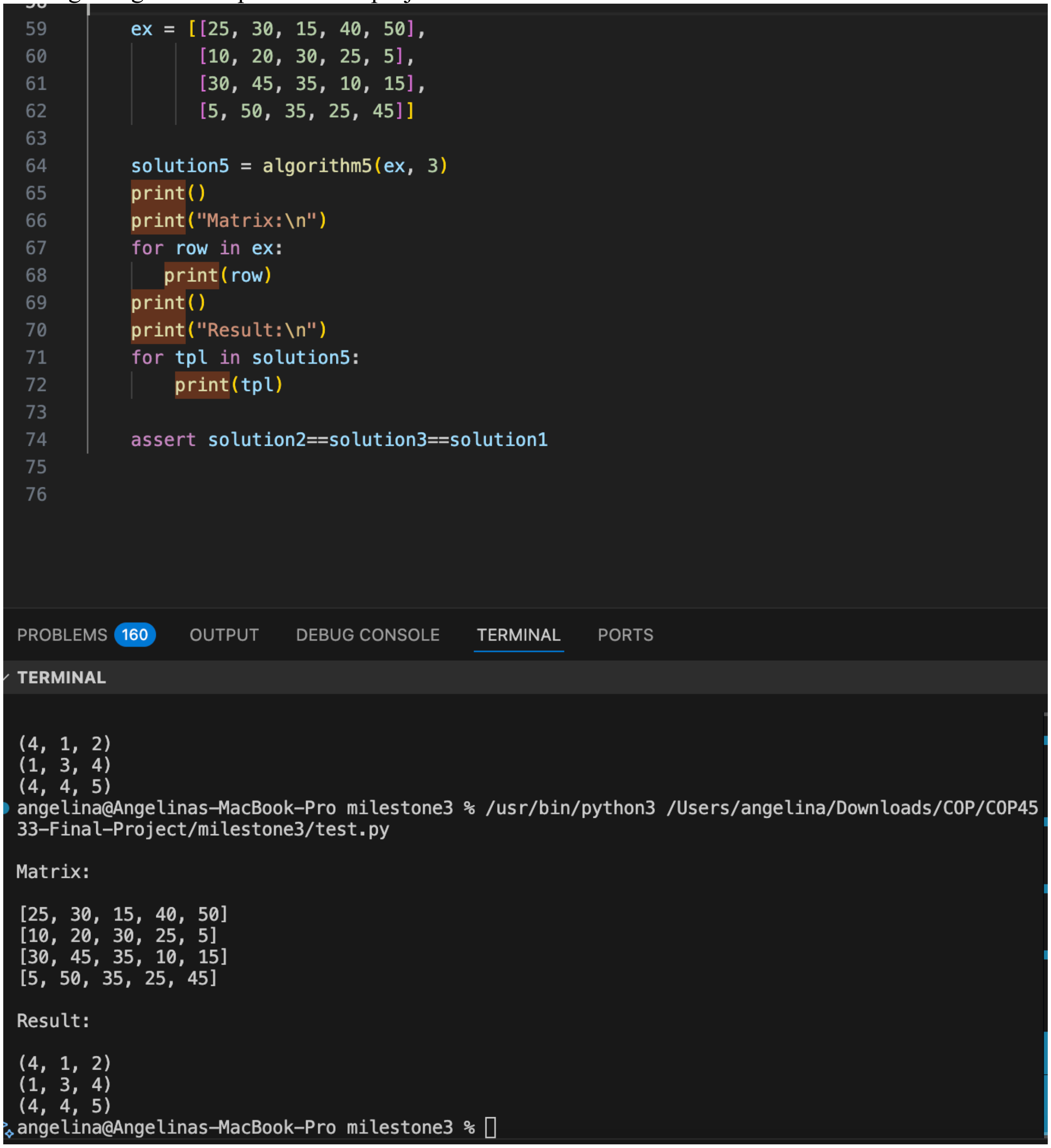
As discussed in the comparative analysis, this approach introduces a significant space-time trade-off. While it improves runtime for large inputs, the storage requirement can be unrealistically high. For small input sizes, a naïve or partially optimized approach would be more efficient in terms of both implementation simplicity and memory usage.
Testing
The current algorithm was rigorously tested using both randomly generated and normally distributed matrices to mimic realistic trading patterns. Over 10 complex test cases were evaluated, and all produced correct and consistent results.

Code

```
def algorithm5 (stocks , kval):

    s =  len(stocks) # number of stocks
    d =  len(stocks[0])  #number of days
    # 3d DP array to store optimal profits where each index(day) represents a potential
sell day for the current stock and transaction count k
    opt = [[[0 for _ in range(kval)] for _ in range(d)] for _ in range(s)]
    # 2d array to track minimum price day (best day to buy) for each stock and
transaction count k
    buy = [[0 for _ in range(kval)]  for _ in range(s)]

    # 3d array with precomputed profits for each stock, buyday, and sellday
    sums = [[[0 for _ in range(d)] for _ in range(d)] for _ in range(s)]

    #2d array to track best sell day for each stock and transaction
    sell = [[0 for _ in range(kval)]for _ in range(s)]


    # Precompute max profit for each buy-sell day, where actual buy day is any day in the
range [buy, sell-1] that produces the highest profit if sold on sell day
    for m in range(s):
        for  i in range(d):
            min_val = float('INF')
            for  j in range( i+1, d):
                min_val  = min(min_val, stocks[m][j-1] )
                sums[m][i][j] = stocks[m][j] - min_val
                if (m > 0 and  sums[m-1][i][j] > stocks[m][j] - min_val ):
                      sums[m][i][j] = sums[m-1][i][j]

    #Build a DP array opt
    for  n in range(1, d): # days

        for  m in range (s): # stocks
            improvement=-1 # track whether adding a new k-th transaction improves the
global profit

            for  k in range(0, kval): #transaction count


                buyday = buy[m][k] #current buyday
                sellday = sell[m][k] #the last sell day from k-1 transaction

                # Propagate the minimum buy day from the previous k-th transaction.
                # If a better (either lower-priced or newly chosen due to adding one more
transaction)
                #  buy day was found at transaction k-1,
                # it should be propagated to all future transactions (k+i) where k+1 <
kval.
                if(k>0):
                    buyday = max(buy[m][k-1], buy[m][k])

                # Case 1: first stock and first transaction. Can reuse a previous optimal
or sell today.
                if(k==0 and m==0):
                     opt[m][n][k]  = max(opt[m][n-1][k], sums[m][buyday][n])

                # Case 2: first transaction. Can reuse a previous optimal, upper row
optimal, or sell today.
                elif(k==0):
                     opt[m][n][k]  = max(  opt[m][n-1][k], sums[m][buyday][n], opt[m-
1][n][k])

               # Case 3:  First row (m == 0), multiple transactions.
               #We can either reuse a previous optimal, or add the best profit to the
optimal profit for k-1 for the current k-th transaction for mth row.
                elif(m==0):
                    opt[m][n][k]  = max( opt[m][n-1][k],
                                    opt[m][n-1][k-1] + sums[m][buyday][n])

                # Case 4: No restrictions.
                # We can either reuse a previous optimal, or add the best profit to the
optimal profit for k-1 for the current k-th transaction.
                # The profit comes from any buy-sell pair where the buy day is meanimum
day after the k-1th sell day.
                # This logic is already handled via the precomputed sums array and the
tracked sellday for each k.
                else:
                    opt[m][n][k]  = max( opt[m-1][n][k],
                                    opt[m][n-1][k],
                                    opt[m][buyday][k-1] + sums[m][buyday][n], #or
s[current] - s[buyday] - it's the same
                                    opt[m][sellday][k-1] +sums[m][sellday][n],
                                    opt[m][n][k-1]
                                    )



                # If the current day's price is lower than the currently tracked minimum
for stock m,
                # update the buy day and propagate it forward for all upcoming days and
stocks.
                if( stocks[m][n] <= stocks[m][buyday]) :
                    buy[m][k] = n

                # A previous result has been reused - selling on the current day gives no
improvement.
                # Skip executing the next block and continue.
                if((opt[m][n][k] == opt[m][n-1][k])  \
                    or (m>0 and opt[m][n][k] == opt[m-1][n][k] ) \
                    or (k>0 and opt[m][n][k] == opt[m][n][k-1])):
                      continue
                # The first improvement has occurred – mark the transaction count index
(k), where the first meaningful profit increase happened
                else:
                   if improvement==-1:
                        improvement = k



                # If selling on the current day results in an improved global profit for
transaction k (improvement == k),
                # then for the next day and the next transaction (n+1, k+1), we should
start tracking from this day
                # as the new potential(minimum price) buy point
            if improvement!=-1 and improvement<kval-1:
                buy[m][improvement+1] = n
                sell[m][improvement+1] = n


    # Find the maximum profit and the number of transactions used
    k=-1
    profit = 0
    # Find the minimum number of transactions (from 0 to k) needed to achieve the maximum
profit
    for  i in range(kval):
        if (opt[s-1][d-1][i] > profit):
            profit = opt[s-1][d-1][i]
            k = i

    #Return early if no profit is possible
    if k==-1:
        return (0, 0, 0)

    #Initialize current sell day and stock number
    sell = d- 1
    stock = s-1


  # Find last sell day (where the optimal values are reused)
    transactions = []
    while sell>0 and k>-1:

        # Same profit without using current stock, move up
        if stock >0 and opt[stock][sell][k] == opt[stock -1][sell][k]:
            stock -=1

        # Same profit as on the previous day, move left
        elif opt[stock][sell][k] == opt[stock][sell-1][k]:
            sell-=1

        #Same profit with fewer transactions
        elif k>0 and opt[stock][sell][k] == opt[stock][sell][k-1]:
            k-=1
        # A sale has occured on the current candidate sell day
        else:

# A new transaction has been used: find the stock i and buy day j
# such that selling at day 'sell' gives the maximum added profit in this range and is
consistent with opt array

            cstock = -1
            cbuy = -1
            found = False
            for  i in range(stock, -1, -1): #  iterate over all stocks up to this point
                for  j in range(sell, -1, -1): # iterate over all buy days up to the
current day
                    current = j

                   # Case 1: If k > 0, the current optimal is formed by:
                    # the previous optimal from the k-1th transaction + the profit from
the best k-th transaction (selling on "sell" day)
                    if k>0:
                        diff = opt[m][sell][k]  - opt[m][current][k-1]

                    # When k = 0 (zero-based), the current optimal corresponds to a
single transaction starting from 0
                    else:
                        diff = opt[m][sell][k]

                    # If the values in the opt array logically match the stock price
differences,
                    # we have found the correct buy day index
                    if diff == stocks[i][sell]- stocks[i][current]:
                            cstock = i #current stock
                            cbuy = current #current buy
                            found = True
                            break
                    if found == True:
                        break

            # Add the current transaction tuple to the list
            transactions.append((cstock+1, cbuy+1, sell+1))
            sell = cbuy
            stock=s-1
            k-=1

# Reverse the order since we backtracked from the end of the array
```

```
    transactions = list(reversed(transactions))


    return transactions
```

Testing using an example from the project file:



```
    ex = [[25, 30, 15, 40, 50],
          [10, 20, 30, 25, 5],
          [30, 45, 35, 10, 15],
          [5, 50, 35, 25, 45]]

    solution5 = algorithm5(ex, 3)
    print()
    print("Matrix:\n")
    for row in ex:
        print(row)
    print()
    print("Result:\n")
    for tpl in solution5:
        print(tpl)

    assert solution2==solution3==solution1
```

PROBLEMS 160 OUTPUT DEBUG CONSOLE TERMINAL PORTS

TERMINAL

```
(4, 1, 2)
(1, 3, 4)
(4, 4, 5)
angelina@Angelinas-MacBook-Pro milestone3 % /usr/bin/python3 /Users/angelina/Downloads/COP/COP45
33-Final-Project/milestone3/test.py

Matrix:

[25, 30, 15, 40, 50]
[10, 20, 30, 25, 5]
[30, 45, 35, 10, 15]
[5, 50, 35, 25, 45]

Result:

(4, 1, 2)
(1, 3, 4)
(4, 4, 5)
angelina@Angelinas-MacBook-Pro milestone3 % 
```

Task 6:

Modifications

There were slight modifications to the original code being around the handling of skipped days during the transactions to ensure that the DP array correctly stores the maximum profit even if there are no transactions occurring. The overall structure and backtracking logic remain the same but this was something that needed to be accounted for and was apparent during testing. The other changes were due to syntax which was expected when implementation the pseudo code logic into python.

Assumptions & Limitations

Assumptions:

- Input 2D matrix A where m is the number of stocks and n is the number of days
- Cool-down period c is a non-negative integer and constraint for the next available buy day
- The output should be a tuple with the best transactions made for maximum profit following these constraints
- If matrix A is empty, we should return an empty list [] same output if there is no profit available

Limitations:

- Runtime is inefficient for larger input sizes since the algorithm runs O(m*n^2)
- Implementation although simpler than the optimized algorithm for task 7, is still complex due to the DP approach
- Larger cool-down period can lead to limitations in maximizing potential profit since it would cause for longer wait times between transactions

Description of Implementation (with Proof)

This algorithm uses a DP approach specifically tabulation to find the maximum profit given the multiple stocks available while also considering the cool-down period between transactions. It first iterates through all possible buy/sell day pairs (j1, j2) for each stock and calculates the potential profits that can be made. If the transaction improves the overall profits compared to the current best profit up to each day, we update the DP array and the transaction array. The DP array stores the maximum profit achievable up to each day.

Once all transactions have been made, we check if the profit for the days skipped is greater than the current profit computed. If it is, we update to the skipped profit and skipped transactions. Finally, we backtrack from the last day using the transactions array to get the best sequence that yields the maximum profit. This approach ensures we find the global optimal solution for the given matrix and its constraints.

Time Complexity

This algorithm has a time complexity of O(m*n^2) where m represents the number of stocks and n represents the number of days. The outer loop iterates through all buy/sell day pairs (j1, j2) which would run O(n^2). For each pair it loops through all stocks resulting in the time complexity of O(m*n^2). Each profit calculation is done in constant time making the overall time complexity for this algorithm O(m*n^2). We backtrack through to get the best sequence for maximum profit which runs in O(n) as we loop backwards through the transactions array to store the best transactions to the results list. The overall time complexity, however, remains the same running O(m*n^2).

The space complexity is O(n) due to the DP array O(n) storing the maximum profit up to each day and the transactions array being used to store the transactions made O(n). Therefore, the overall space complexity is O(2n) which simplifies to O(n).

Comparative Analysis with Other Implementations

This algorithm is easier to implement but is less efficient than the optimized O(m*n) algorithm in task 7. As the input size of n increases this algorithm will become slower making it less optimal for larger datasets. The optimized algorithm in task 7 uses a better strategy that avoids checking all pairs, which would improve the runtime at the cost of complexity. Our solution checks all possible pairs of days for each stock which is more time consuming than the approach taking in task 7. Compared to the single stock transactions like in task 1 or 2, these algorithms are more complex but also have different constraints such as multiple transactions being allowed, and a cool-down period that needs to be tracked.

Trade-off Discussion

Although this algorithm is simpler to implement, it is slower than the task 7 optimized algorithm. This wouldn't be a problem for smaller input sizes, but as n grows so downs the time needed to compute the solution. The optimized algorithm would be more complex to implement but would be better suited for larger input sizes and would scale better as n increases. Both algorithms use O(n) space because of the DP approach, which will take more memory.

Code

```
"""
Task 6 Code Implementation
DP Solution approach for stock trading with a cool-down period c
Input: 2D matrix A and c representing the cool-down period
Output: We return a list of tuples representing the best transactions made for max profit
Run time O(m*n^2) where m is the number of stocks and n is the number of days
"""

def algorithm6(A, c):
    # if the array is empty, or there are no stocks, or no days, return empty list []
    if not A or len(A) == 0 or len(A[0]) == 0:
        return []

    m = len(A)
    n = len(A[0])

    #if m == 0 or n == 0:
    #   return []

    # store best profit up to the each day
    dp = [0] * n
```

```python
    # store transactions that give best profit of day
    transactions = [None] * n

    # intialize results to store optimal transaction sequence
    results = []
    # initialize current day to the last day for backtracking
    currDay = n - 1

    # go through all buy and sell day pairs
    for j1 in range(0, n - 1):
        for j2 in range(j1 + 1, n):
            # iterate through each stock
            for i in range(m):
                # calculate profit sell day - buy day
                profit = A[i][j2] - A[i][j1]
                if profit < 0:
                    # skip transaction if there is no profit
                    continue

                # check for previous day after cool-down
                prevDay = j1 - c -  1
                if prevDay >= 0:
                    totalProfit = profit + dp[prevDay]
                else:
                    totalProfit = profit

                # update best profit and store transaction
                if totalProfit > dp[j2]:
                    dp[j2] = totalProfit
                    transactions[j2] = (i, j1, j2, prevDay)

    # Check skipped days with best profits
    for skippedDays in range(1,n):
        if dp[skippedDays-1] > dp[skippedDays]:
            dp[skippedDays] = dp[skippedDays-1]
            # only get transaction if current one is none existent
            if transactions[skippedDays] is None:
                transactions[skippedDays] = transactions[skippedDays-1]


    # backtrack to get best sequence
    while currDay >= 0:
        if transactions[currDay] is not None:
            i, j1, j2, prevDay = transactions[currDay]
            # Get 1-based index for output
            results.append((i+1, j1+ 1, j2 + 1))
            # go to the previous transaction day
            currDay = prevDay
        else:
            # go to previous day
            currDay -= 1

    # reverse to get correct order output earliest transaction first
    results.reverse()
    return results
```

Task 7:

Modifications

Assumptions & Limitations

The problem does not list that return of an empty tuple is necessary when no profitable trades are found. I ignored that as a potential approach. All potential trades are saved and then we backtrack using the timeframe, c, specified. The length of the list holding the decisions was changed from n to m as testing with non-square matrices would crash the program

Description of Implementation (with Proof)

This approach loops through days and then stocks. Reviewing each stock of a given day before moving to the next. This is by the far the largest growth in the algorithm and using our typical asymptotic assumptions we arrive at the algorithm running, O(m*n).

We know that we need to prove that not only do we track the most profitable trade, but that we need to ensure we only care about the best trade within cooldown.

We traverse M stocks, storing the lowest price we find for each. As we move down n days we keep tracking lowest price and also highest price to determine an advantageous trade. We notate that trade and keep moving. When we do so, we can ensure that cooldown is respected by storing the date.

We then look back through the returned tuples to ensure that only the most advantageous trades are kept. This occurs in linear time and as such can be ignored.

Time Complexity

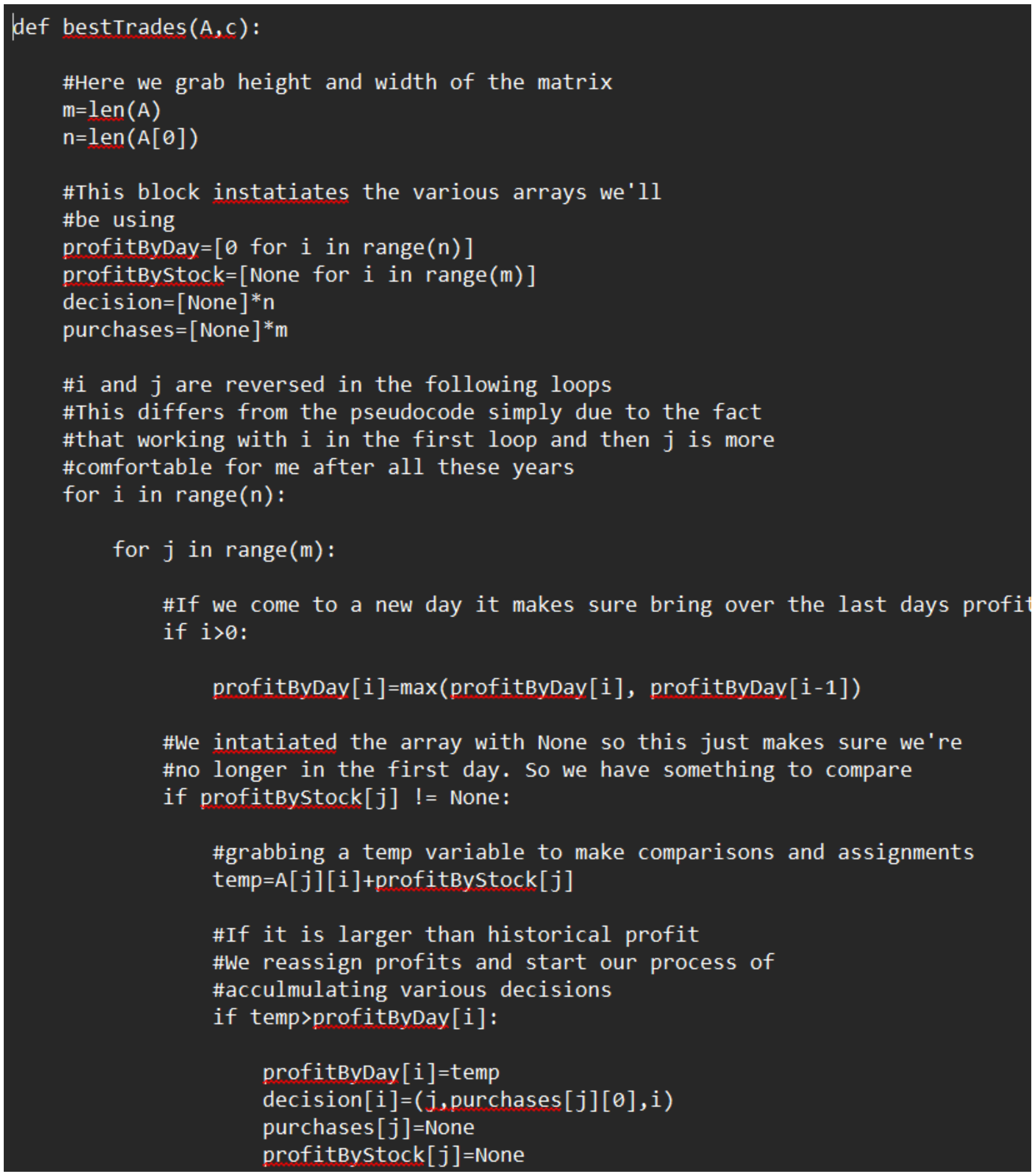
O(m*n)

Comparative Analysis with Other Implementations

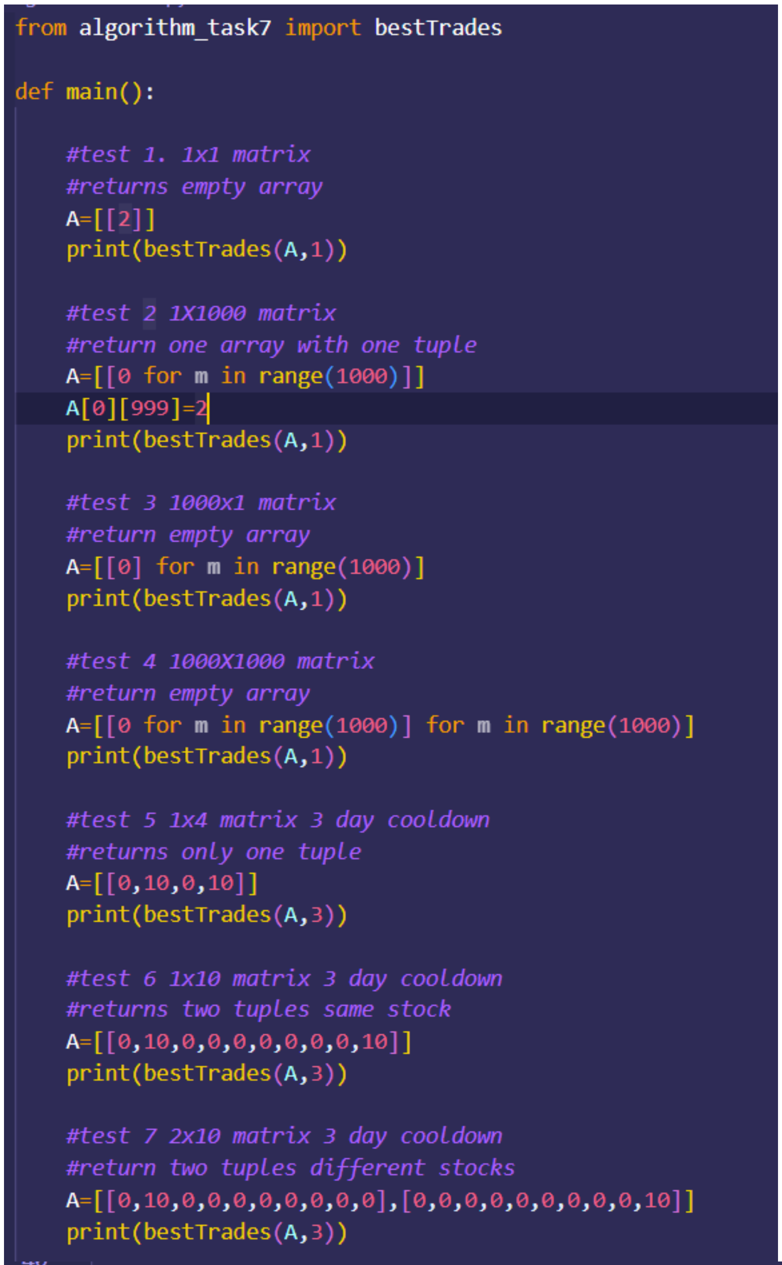
The major factor for this algorithm is that utilizing dynamic programming to save the potential profit by each day allows for the algorithm to traverse m first and then n. Resulting in visiting each element of the matrix only once which is significantly quicker than task 6 where the O(m*n^2) is much slower than task 7's O(m*n)

Trade-off Discussion

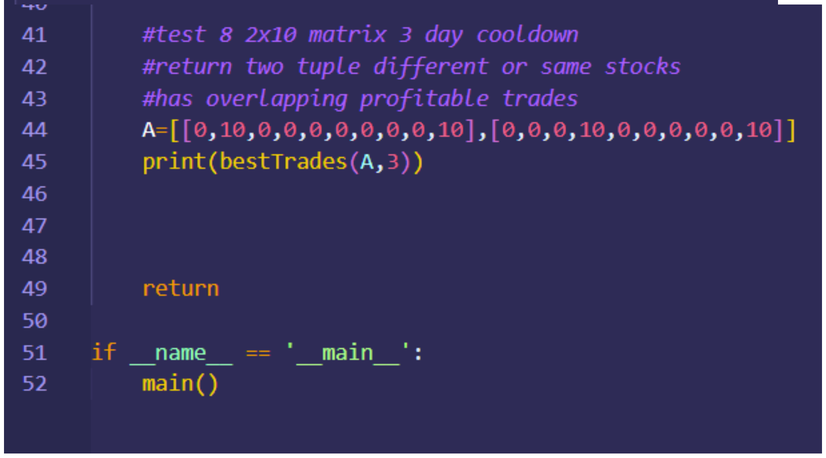
This algorithm obviously has greater space complexity than other approaches. It's a rather minor trade off for any modern system.



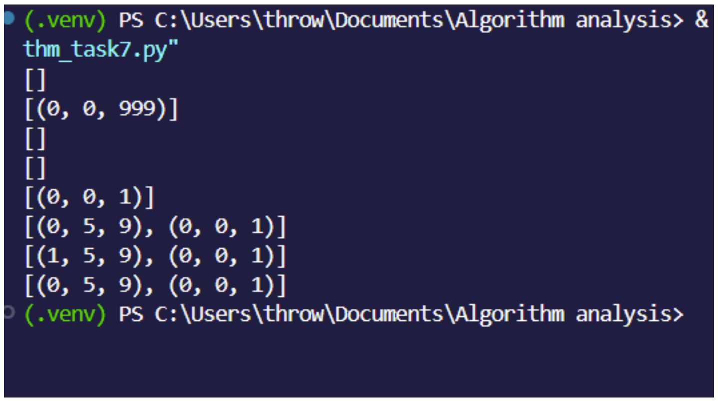
```
def bestTrades(A,c):

    #Here we grab height and width of the matrix
    m=len(A)
    n=len(A[0])

    #This block instatiates the various arrays we'll
    #be using
    profitByDay=[0 for i in range(n)]
    profitByStock=[None for i in range(m)]
    decision=[None]*n
    purchases=[None]*m

    #i and j are reversed in the following loops
    #This differs from the pseudocode simply due to the fact
    #that working with i in the first loop and then j is more
    #comfortable for me after all these years
    for i in range(n):

        for j in range(m):

            #If we come to a new day it makes sure bring over the last days profit
            if i>0:

                profitByDay[i]=max(profitByDay[i], profitByDay[i-1])

            #We intatiated the array with None so this just makes sure we're
            #no longer in the first day. So we have something to compare
            if profitByStock[j] != None:

                #grabbing a temp variable to make comparisons and assignments
                temp=A[j][i]+profitByStock[j]

                #If it is larger than historical profit
                #We reassign profits and start our process of
                #acculmulating various decisions
                if temp>profitByDay[i]:

                    profitByDay[i]=temp
                    decision[i]=(j,purchases[j][0],i)
                    purchases[j]=None
                    profitByStock[j]=None
            #This following if else block does two things
            #first to checks if we're past a certain number of days
            #then checks back to see what the stock price would have been and
            #assigns it if its better than holding

            #The second portion is just how we first approach the problem if we're not a
            #certain number of days out
            if i-c-1>=0:

                profit = profitByDay[i-c-1]-A[j][i]

                if profitByStock[j] == None or profitByStock[j]<profit:
                    profitByStock[j]=profit
                    purchases[j]=(i,A[j][i])

            elif i-c-1 < 0:
                profit = -A[j][i]
                if profitByStock[j] == None or profitByStock[j]<profit:
                    profitByStock[j]=profit
                    purchases[j]=(i,A[j][i])


    #This creates the array we're actually going to return
    tuplesToReturn=[]
    i=n-1

    #We run backwards in time through the decision array. jumping backward over the cooldown period because
    #we are grabbing everything in the main algorithm and need to keep that mind
    while i>=0:
        #print(n)
        #print(i,decision[i])
        if decision[i] != None:
            tuplesToReturn.append((decision[i][0],decision[i][1],decision[i][2]))
            i=decision[i][1]-c-1
        else:
            i=i-1


    #return the tuples
    return tuplesToReturn
```



```python
from algorithm_task7 import bestTrades

def main():

    #test 1. 1x1 matrix
    #returns empty array
    A=[[2]]
    print(bestTrades(A,1))

    #test 2 1X1000 matrix
    #return one array with one tuple
    A=[[0 for m in range(1000)]]
    A[0][999]=2
    print(bestTrades(A,1))

    #test 3 1000x1 matrix
    #return empty array
    A=[[0] for m in range(1000)]
    print(bestTrades(A,1))

    #test 4 1000X1000 matrix
    #return empty array
    A=[[0 for m in range(1000)] for m in range(1000)]
    print(bestTrades(A,1))

    #test 5 1x4 matrix 3 day cooldown
    #returns only one tuple
    A=[[0,10,0,10]]
    print(bestTrades(A,3))

    #test 6 1x10 matrix 3 day cooldown
    #returns two tuples same stock
    A=[[0,10,0,0,0,0,0,0,0,10]]
    print(bestTrades(A,3))

    #test 7 2x10 matrix 3 day cooldown
    #return two tuples different stocks
    A=[[0,10,0,0,0,0,0,0,0,0],[0,0,0,0,0,0,0,0,0,10]]
    print(bestTrades(A,3))
```



```python
    #test 8 2x10 matrix 3 day cooldown
    #return two tuple different or same stocks
    #has overlapping profitable trades
    A=[[0,10,0,0,0,0,0,0,0,10],[0,0,0,10,0,0,0,0,0,10]]
    print(bestTrades(A,3))



    return

if __name__ == '__main__':
    main()
```



```
(.venv) PS C:\Users\throw\Documents\Algorithm analysis> &
thm_task7.py"
[]
[(0, 0, 999)]
[]
[]
[(0, 0, 1)]
[(0, 5, 9), (0, 0, 1)]
[(1, 5, 9), (0, 0, 1)]
[(0, 5, 9), (0, 0, 1)]
(.venv) PS C:\Users\throw\Documents\Algorithm analysis>
```

Conclusion

Lessons Learned

Angelina:
I have expanded my understanding of the dynamic programming approach. It used to be a very challenging domain of algorithms for me, and even the simplest tasks would take a considerable amount of time to work through. Although it took many days to finally arrive at solutions for more complex problems, I was able to grasp the underlying structure they commonly share. I now understand much better how to approach such problems from the beginning. I realized that working out an example step-by-step, documenting every observation and assumption, is the best way to start formulating the recurrence relation.

Jacob:
This final project helped me better understand why different approaches are necessary when it comes to solving a similar problem. I struggled with dynamic programming and was able to dive deeper into understanding its advantages and its limitations. Writing pseudocode and documenting my logic saved me from a lot of frustration and time wasted. It allowed me to visualize the issues and think of the potential problems with my approach. I had worked on task 4 and was having difficulties in getting the correct time complexity, which could had probably been more attainable if I had a better approach or understanding prior.

Group:
Overall, as a team, we learned to collectively brainstorm the problem and share our observations, which helped us arrive at a solution more quickly. It was a unique and valuable experience working on a new type of problem that focused on non-trivial algorithmic thinking rather than following fixed steps.

Limitations of the Algorithms

All limitations have been discussed individually for each algorithm. However, starting from Task 4, the main limitation shared by the more complex algorithms is their runtime complexity, which makes them unsuitable for very large input sizes.

Analysis of the Algorithms

All algorithms inherently rely on a common structure where the global optimum can be broken down into smaller recursive components. By leveraging this property, the overall time complexity can be significantly reduced compared to brute-force approaches.