# 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

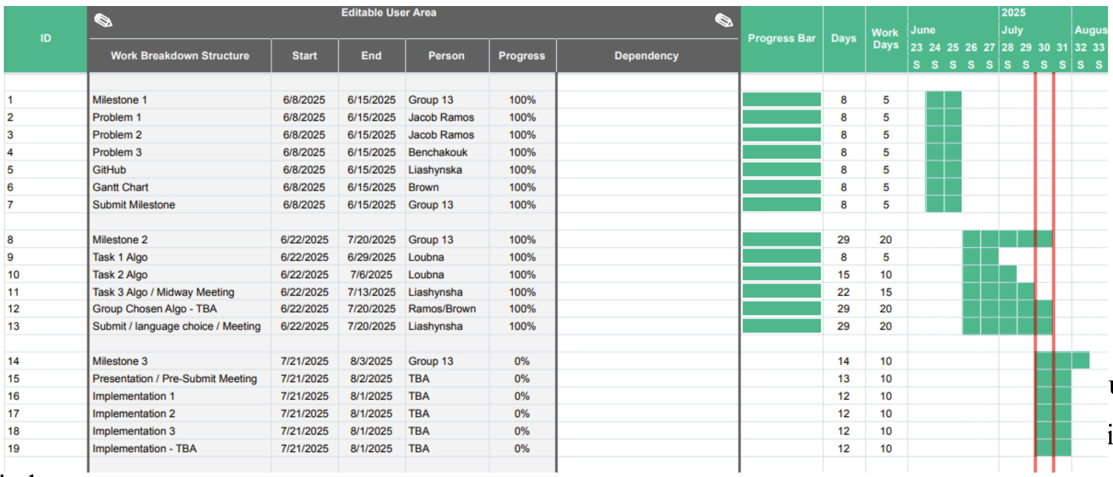
**Member Roles:**

- **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 | | | | Augus | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Work Breakdown Structure | Start | End | Person | Progress | Dependency | | | | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 | 32 | 33 |
| | | | | | | | | | | S | S | S | S | S | S | S | S | S | S | 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

**Language decision**:
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][$j_1$]

```
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)
```

**Pseudocode:**

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 $O(m \cdot n)$

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 $O(m \cdot n^2)$

**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.