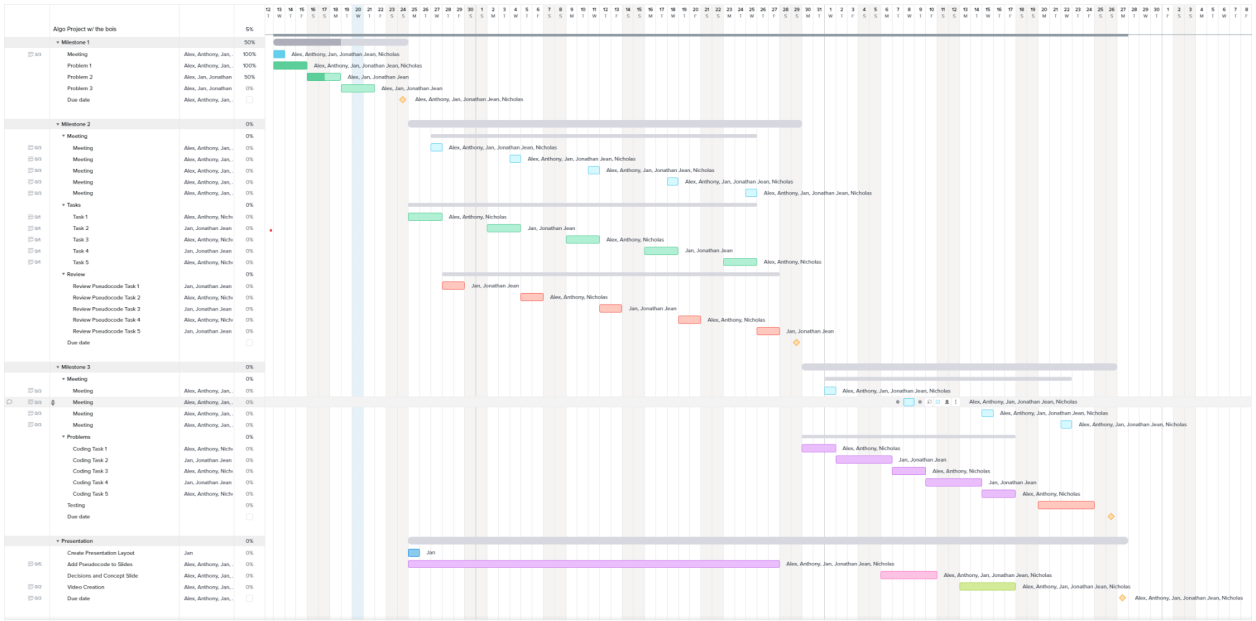
Project - Milestone 1

Group Members:

- Jan Torruellas
- Alex Tran
- Huan (Nicholas) Tran
- Jonathan Jean
- Anthony Gravier

GitHub Link:
https://github.com/jtorruellas22/Algorithms-Abstraction-Design-Project/commits/main

Gantt Link:
https://app.teamgantt.com/projects/gantt?ids=3668406



Communication Method:
Discord

Roles:

Project Manager: Jan Torruellas
Gantt Manager: Jonathan Jean
Developer: Alex Tran
Developer: Nicholas Tran
Developer: Anthony Gravier

4.1: Problem 1

For stock with index 1:

$$A = \begin{bmatrix} 12 & 1 & 5 & 3 & 16 \\ 4 & 4 & 13 & 4 & 9 \\ 6 & 8 & 6 & 1 & 2 \\ 14 & 3 & 4 & 8 & 10 \end{bmatrix}$$

- Given buying on day 1: (1, 1, 2,-11), (1, 1, 3, -7,), (1, 1, 4, -9), (1, 1, 5, 4)
- Given buying on day 2: (1, 2, 3, 4), (1, 2, 4, 2), (1, 2, 5, 15)
- Given buying on day 3: (1, 3, 4, -2), (1, 3, 5, 11)
- Given buying on day 4: (1, 4, 5, 13)

For stock with index 2:

- Buying on day 1: (2, 1, 2, 0), (2, 1, 3, 9), (2, 1, 4, 0), (2, 1, 5, 5)
- Buying on day 2: (2, 2, 3, 9), (2, 2, 4, 0), (2, 2, 5, 5)
- Buying on day 3: (2, 3, 4, -9), (2, 3, 5, -4)
- Buying on day 4: (2, 4, 5, 5)

For stock with index 3:

- Buying on day 1: (3, 1, 2, 2), (3, 1, 3, 0), (3, 1, 4, -5), (3, 1, 5, -4)
- Buying on day 2: (3, 2, 3, -2), (3, 2, 4, -7), (3, 2, 5, -6)
- Buying on day 3: (3, 3, 4, -5), (3, 3, 5, -4)
- Buying on day 4: (3, 4, 5, 1)

For stock with index 4:

- Buying on day 1: (4, 1, 2, -11), (4, 1, 3, -10), (4, 1, 4, -9), (4, 1, 5, 4)
- Buying on day 2: (4, 2, 3, 1), (4, 2, 4, 5), (4, 2, 5, 7)
- Buying on day 3: (4, 3, 4, 4), (4, 3, 5, 5)
- Buying on day 4: (4, 4, 5, 2)

Step 3.

- For stock 1, the day with the highest potential profit is day 2.
- For stock 2, the day with the highest potential profit is day 1 or day 2.
- For stock 3, the day with the highest potential profit is day 1.
- For stock 3, the day with the highest potential profit is day 2.

Step 4.

- The stock and day combination that yields the maximum potential profit is (1, 2, 5, 15).

4.2 Problem 2:

Given Matrix:

$$A = \begin{bmatrix} 25 & 30 & 15 & 40 & 50 \\ 10 & 20 & 30 & 25 & 5 \\ 30 & 45 & 35 & 10 & 15 \\ 5 & 50 & 35 & 25 & 45 \end{bmatrix}$$

Answer for k = 3:
Analysis:
Step 1: Buy 4th stock on day 1, sell on day 2
Step 2: Buy 2nd stock on the 2nd day, sell on the 3rd day
Step 3: Buy 1st stock on 3rd day, sell on 5th day
Total profit: 50 - 5 = 45
30 - 20 = 10
50 - 15 = 35
= 90
Output: [(4,1,2), (2,2 3), (1,5)], k = 3 transactions

4.3 Problem 3:

Given Matrix:

$$A = \begin{bmatrix} 7 & 1 & 5 & 3 & 6 & 8 & 9 \\ 2 & 4 & 3 & 7 & 9 & 1 & 8 \\ 5 & 8 & 9 & 1 & 2 & 3 & 10 \\ 9 & 3 & 4 & 8 & 7 & 4 & 1 \\ 3 & 1 & 5 & 8 & 9 & 6 & 4 \end{bmatrix}$$

Answer for c = 2:
Analysis:

Step 1: Buy 3rd stock on day 1, sell on day 3
Step 2: Buy 2nd stock on day 6, sell on day 7

Total Profit:
Profit from step 1: (9 - 5) = 4

Profit from step 2: (8 - 1) = 7

Total Profit: (4+7) = 11

Output = [(3,1, 3), (2,6,7)]

Project - Milestone 2

Languages we will be using to implement the below pseudocode:

Java
C++

Task 1 - Pseudocode

Given a Matrix A, profit p = 0, max = 0, buy b, sell s, stock x, bought y, sold z

Given a list of stocks, that each have their own list of prices.
Ex: [[1, 2, 6, 8], [2, 4, 8, 9]]

[1 2 6 8]
[2 4 8 9]

```
// n is the number of stocks that are inside of the matrix, or 2D array.
// m is the number of days that the stock has recorded prices for it.
Func processMaxTransaction() {
        Int maxPrice = 0;
        Int stock = 0;
        Int boughtDay = 0;
        Int soldDay = 0;
        For (int stock = 0; stock < stocks.size(); stock++) {   // O(n)
                For (int i = 0; i < stocks.get(i).size(); i++) {  // O(m)
                        If (i == stocks.get(i).size() -1) {
                                break;
                        }
                        For (int j = i+1; j < stocks.get(i).size(); j++) { // O(m)
                                Int priceForDay = sp.at(j) - sp.at(i);
                                If ( priceForDay > maxPrice ){
                                        maxPrice = priceForDay;
                                        stock = stock;
                                        boughtDay = i;
                                        soldDay = j;
                                }
                        }
                }
        }
        Return (stock, boughtDay, soldDay, maxPrice);
}
```

Task 2- Pseudocode

Given a Matrix A, profit p = 0, min = 0, max = 0, buy b, sell s, stock x, bought y, sold z

```
// n is the number of stocks that are inside of the matrix, or 2D array.
// m is the number of days that the stock has recorded prices for it.

For stock m in A
        For day n in m
                // On the first day we attempt a buy
                If n is first day:
                        Min = A[m][n]
                        b = n
                // Compare if the buy amount is less than our current buy amount and update
                Else if A[m][n] < min
                        Min = A[m][n]
                        b = n
                        Max = 0
                // If selling on a given day yields a greater profit we sell and update our values
                Else if A[m][n] - min > max:
                        Max = A[m][n]
                        s = n
                        If (p < max - min):
                                p = max - min
                                x = m
                                y = b
                                z = s
Return (x, y, z, p)
```

Each day, the greedy choice is being made to find the cheapest day to buy. Each time the cheapest day to buy is found, the current maximum price to sell at is reset to 0, due to it being impossible to buy and sell the same day or previous days. Once a day to sell is found, the profit that can be made in this current “window” is compared to the most profit we have made so far. If it is greater, then the most profit made so far p is updated with this new profit. The “window” that generated the highest profit so far is also stored in y (day bought) and z (day sold). The stock that corresponds to this buy/sell window is stored in x. At the end of the function x,y,z, and p are returned as a tuple.

Task 3 - Pseudocode

```
M = price of stocks, n = day

Func MaxProfit(stockMatrix[m][n)
{
        If m < 2 or n < 2 return 0
        //cannot make a transaction if value of the prices or number of days is less than one
        Int minPrice = array[n]
        Int maxProfit = array[n]
        minPrice[0] = stockMatrix[0][0] //
        Maxprofit = 0

        //initialize min price array, m = price of stocks, n = day

        // Initialize the first element of min_price and max_profit
        min_price[0] = A[0][0]
        max_profit[0] = 0  // No profit on the first day

         // Initialize min_price array
         for i from 1 to n-1:
          min_price[i] = min(min_price[i-1], A[0][i])

        // Calculate maximum profit for each day
        for i from 1 to m-1:
        for j from 0 to n-1:
        // Calculate the profit if we sell on day j and bought on the day with minimum price
        max_profit[j] = max(max_profit[j], A[i][j] - min_price[j])

        // Update the minimum price if we find a lower price on this day
        min_price[j] = min(min_price[j], A[i][j])

   // Find and return the maximum profit
   return max(max_profit)

}
```

Task 5 - Pseudocode

DP algorithm for Problem 2:

Ex: [1, 6, 8, 3]
[2, 5, 7, 9]
[12, 2, 6, 3]

Given a matrix array A of m x n, and k meaning max number of transactions, find the maxProfit in the array given at most k transactions.

```
func MaxProfit(A, k) {
        M, n = dimensions of A
        Max_prof = arr[n]

        //Case where m < 2, n <2
        If m < 2 or n < 2 return 0 //

        //3D array
          for i from 0 to m-1:
              for j from 0 to n-1:
        //No profit with 0 transactions
        Max_prof[i][j][0] = 0
        For t from 1 to k:
        //Initialize a large negative value
        Max_prof[i][j][t] = -infinity
        // Calculate maximum profit for each day and each transaction count
            for i from 0 to m-1:
                for t from 1 to k:
                    for j from 1 to n-1:
                        // Calculate the maximum profit for the current day and transaction count
                        Max_prof[i][j][t] = max(
                            //Maximum profit without making a transaction on day j
                            Max_prof[i][j-1][t],
                             //Maximum profit from the previous day with the same transaction count
                            Max_prof[i-1][j][t],
                            //Maximum profit by selling on day j and buying on the previous day
                            Max_prof[i-1][j-1][t-1] + A[i][j] - A[i-1][j] )

            // Return the maximum profit with k transactions
            return Max_prof[m-1][n-1][k]


}
```

Task 7 - Pseudocode

Given:
map P, map memTable, Int c, List intervals

/*The list of intervals is assumed to be sorted by sell day

/* Each interval in the list of intervals has the following properties:
(Int product, Int buyDay, Int sellDay, Int profit) */

/* P is a map from integers to pairs containing an interval and another integer.
The idea is that for a given key n in P that represents the index of an interval in the list of intervals, it will map to a pair containing the closest interval that comes before interval n in the list of intervals mentioned above, along with an integer that is <= n - 1 that represents the index of the closest interval in the list of intervals that does not overlap. */

/* memTable is a map from integers to integers.
The idea is that for a given key k in memTable that represents the index of an interval in the list of intervals, k will map to the maximum profit that can be made at index k in the sorted list of intervals. */

// Int c represents the number of days that we must wait to purchase a stock after selling a stock.

PseudoCode:
//This function outputs the list of intervals (as a list of tuples) that yield the maximum profit based on the cooldown time c for purchasing stocks
OutputOptimalSolution(P, memTable, c, intervals) {

```
	for (int i = 0; i < intervals.size(); i++) {
		initializeP(p, i, intervals, c);
	}
	int maxProfit = opt(intervals.size() - 1, intervals, memTable, p);
	findSolution(intervals.size() - 1, memTable, intervals, p, output);

	for (int i = output.size()-1; i >= 0; i--) {
		if(i!=0)
			cout << "(" << output[i].product << ", " << output[i].buyDay << ", " <<
output[i].sellDay << "), ";
		else
			cout << "(" << output[i].product << ", " << output[i].buyDay << ", " <<
output[i].sellDay << ")";
```

```
   }
}


//This function initializes the map p. The what each key value pair of the map represents is detailed above.
Void InitializeP(p, intervalIndex, intervals, c) {
	// If the interval at intervalIndex is at the beginning of the list of the intervals, it is impossible for any interval to come before it.
	If (intervalIndex == 0)
	{
		return;
	}

	//For each interval, starting from the interval right before the interval at intervalIndex (which we will call intervalToCalculatePFor in the for loop below) in the sorted interval list…
	For (i = intervalIndex - 1; i >= 0; i–) {
		intervalToCalculatePFor = intervals[intervalIndex];
		currentInterval = intervals[i];
	//Determine if the current intervals' sell day begins more than c days before the buy day of the intervalToCalculatePFor.
		If (currentInterval.sellDay < intervalToCalculatePFor.buyDay - c)
		{
			//If so, map the index of the of intervalToCalculatePFor to the current interval and its index in the list of intervals
			Create newPair;
			newPair.first = currentInterval;
			newPair.second = i;
			P[intervalIndex] = newPair;
			return;
		}
	}
}
//Return the index of the closest interval that comes before the interval at intervalIndex. If there is no key value pair, return -1.
Int GetClosestInterval(intervalIndex,  p) {
	// If no interval is found return -1
	If (not find interval index in p)
	{
		Return -1;
	}

	// Return closest interval
	Return P[intervalIndex].second;
```

```
}

//This function returns that maximum profit that can be made by either including the interval at intervalIndex into our optimal solution, or leaving it out.
Int maxProfit Opt(intervalIndex, intervals,  memTable,  p) {
        //If the interval index is less than 0, than it does not exist and no profit can be made.
        If (intervalIndex < 0)
        {
                Return  0;
        }
        //If we have already calculated the maximum profit that can be made by either including or excluding this interval, then return profit mapped to this index.
        If (memTable[intervalIndex])
        {
                Return memTable[index];
        }
        //Otherwise

        Else
        {
        // Calculate the maximum between the following choices: the profit that can be made by adding the profit of this interval to the profit that can be made at the previous closest interval OR the profit that can be made by at the previous interval in the sorted list of intervals.

                maximum = max(intervals[intervalIndex].profit + opt(getClosestIntervalIndex(intervalIndex, p), intervals, memTable, p), opt(intervalIndex - 1, intervals, memTable, p));

        //Store this maximum in the memoization table so that it does not need to be calculated again for this intervalIndex
                 memTable[intervalIndex] = maximum;

                 return maximum;
        }
}

//This function outputs the intervals that are apart of the optimal solution that yields the maximum profit calculated earlier
findSolution(index, memTable, intervals, p, output) {
        // findSolution is complete, end recursion
        //Base case
        if( index == -1)
                Return;
        Else:
```

```
        // Set the current interval
        currentInterval = intervals[index];

        // Calculate the profit of the current interval plus the closest interval and compare
        to next interval. If so add to output. If not continue to the next interval.
        if(currentInterval.Profit + memTable[getClosestIntervalIndex(index, p)]
                                                        >= memTable[index - 1]) {
                output.push_back(currentInterval);
                findSolution(getClosestIntervalIndex(index, p), memTable, intervals, p,
                                                                output)
        Else
                findSolution(index-1,memTable,intervals, p, output);



}
```

Milestone 3: Code Implementation

Description:

- **The algorithm below begins by iterating over the different stocks, and then iterates over each stock's prices starting at the front, and let us call that the buy price. Next, we start at one day in front of the buy price, and lets call that the sell price. We then subtract the sell price with the buy price, and check a variable called maxPrice, and see if the value we got is greater than the maxPrice.**

Limitations:

- **The algorithm is horrifically slow, leading me to believe that it is O(n*m2), where m is the number of prices that a stock has, and n being the stocks.**

Trade-offs:

- **While the algorithm is extremely slow, it was very easy to solve and write, so if you need a quick solution, and you are analyzing small samples, then this works for you. Anything above 1,000 samples, and this algorithm is not recommended.**

Lessons Learned:

- **While brute-forcing this solution, I found that it helped me think of ways that this problem could be solved with a better time complexity.**

Task 1 Java Implementation:

```
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

public class App {

public static List<Integer> processMaxTransaction(List<List<Integer>>
stocks){
   int maxPrice = 0;
   int stock = 0;
   int boughtDay = 0;
   int soldDay = 0;
   // O(n)
   for (int currentStock = 0; currentStock < stocks.size();
currentStock++) {
```

```
            O(m)
             for (int i = 0; i < stocks.get(currentStock).size(); i++) {
                 if (i == stocks.get(currentStock).size() - 1) {
                     break;
                 }
                // O(m)
                 for (int j = i+1; j < stocks.get(currentStock).size();
j++){
                    int priceForDay = stocks.get(currentStock).get(j) -
                        stocks.get(currentStock).get(i);

                     if ( priceForDay > maxPrice ) {
                         maxPrice = priceForDay;
                         stock = currentStock;
                         boughtDay = i;
                         soldDay = j;
                     }
                 }
             }
         }
         return Arrays.asList(stock, boughtDay, soldDay, maxPrice);
     }

     public static void main(String[] args) throws Exception {
         List<List<Integer>> stocks = new ArrayList<>();
         stocks.add(Arrays.asList(1, 2, 4, 6, 8));
         stocks.add(Arrays.asList(2, 4, 8, 9));

         List<Integer> maxProfit = processMaxTransaction(stocks);
         System.out.printf("the max Profit that can be made is: %d \n",
                  maxProfit.get(3));
         System.out.printf("the stock that was bought is stock: %d \n",
                  maxProfit.get(0) + 1);
         System.out.printf("The stock was bought on day: %d \n",
                  maxProfit.get(1) + 1);
         System.out.printf("The stock was sold on day: %d \n",
                  maxProfit.get(2) + 1);
     }
}
```

Task 2 C++ Implementation:
Description: This solution implements the O(MN) solution for task 2 which is to return the sequence of buy and sell that has the greatest profit.

Limitations:
As there may be many plausible solutions or this simple question may have more than one answer depending on the given matrix, we return a singular solution instead of all solutions.

Trade offs:
This algorithm can only return a single answer to a question which may contain multiple. This algorithm however, will always return a correct answer.

Lessons Learned:
This approach is better than brute force algorithm.

```
#include <iostream>
using namespace std;
void task2 () {
    int profit = 0;
    int min = 0;
    int max = 0;
    int buy, sell, stock, bought, sold;
    int matrix[4][5] = { {12, 1, 5, 3, 16},
                         {4, 4, 13, 4, 9},
                         {6, 8, 6, 1, 2},
                         {14, 3, 4, 8, 10} };
    for (int m = 0; m < 4; m++) {
        for (int n = 0; n < 5; n++) {
            if (n == 0) {
                min = matrix[m][n];
                buy = n;
            }
            else if (matrix[m][n] < min) {
                min = matrix[m][n];
                buy = n;
                max = 0;
            }
            else if (matrix[m][n] > max) {
                max = matrix[m][n];
```

```
            sell = n;
            if(profit < max - min) {
                profit = max - min;
                stock = m;
                bought = buy;
                sold = sell;
            }
        }
    }
    cout << "(" << stock + 1 << "," << bought + 1 << ","<< sold + 1 << ","<< profit << ")" <<
endl;
}

int main() {
    task2();
};
```

Task 3 C++ Implementation (Kadane's algorithm):

Description: This solution implements the dynamic programming pseudocode we developed for task 3 in milestone 2, where given a matrix A of m x n dimensions, we determine the maximum profit possible with a single transaction in O(m*n) running time.

Limitations: The space complexity of this program can get very big if m or n or both are large.

Trade-offs: This algorithm will be more consistent than a greedy algorithm, but a greedy approach might be more optimal if m and n are very small

Lessons learned: This approach is better than brute force approach but may not be better than a greedy algorithm in all cases.

```
#include <iostream>
#include <vector>
#include <tuple>

using namespace std;

tuple<int, int, int, int> maxTransact(vector<vector<int>>& A) {
        int m = A.size();
        if (m == 0) {
        return make_tuple(0, 0, 0, 0); // No stocks available
        }
        int n = A[0].size();
        if (n < 2) {
        return make_tuple(0, 0, 0, 0); // Cannot make any transaction with less than two
days
        }

        int min_price = A[0][0];
        int max_profit = 0;
        int buy_day = 0;
        int sell_day = 0;
        int stock_index = 0;
```

```
        for (int i = 0; i < m; ++i) {
        for (int j = 0; j < n; ++j) {
        if (A[i][j] < min_price) {
                min_price = A[i][j];
                buy_day = j+1; // Update buy_day when a lower price is found
        } else if (A[i][j] - min_price > max_profit) {
                max_profit = A[i][j] - min_price;
                sell_day = j+1; // Update sell_day when a better profit is found
                stock_index = i+1;
        }
        }
        }

        return make_tuple(stock_index, buy_day, sell_day, max_profit);
}

int main() {
        vector<vector<int>> A = {
        {12, 1, 5, 3, 16},
        {4, 4, 13, 4, 9},
        {6, 8, 6, 1, 2},
        {14, 3, 4, 8, 10}
        };

        tuple<int, int, int, int> result = maxTransact(A);
        cout << "Stock Index: " << get<0>(result) << endl;
        cout << "Day Purchased: " << get<1>(result) << endl;
        cout << "Day Sold: " << get<2>(result) << endl;
        cout << "Profit: " << get<3>(result) << endl;

        return 0;
}
```

Task 5 C++ Implementation
Description: This solution implements the dynamic programming pseudocode we developed for task 5 in milestone 2, where given a matrix A of m x n dimensions, we determine the maximum profit possible with a single transaction in O(m*n*k) running time.
Limitations: Some limitations of our algorithm include the time complexity, O(m*n*k), which may be impractical if m, n, and k are large. Also, the algorithm may not always find the globally optimal solution, since it relies on dynamic programming to make decisions based on previous states. Finally, the algorithm is designed for buying and selling single stocks at a time; it would require some modifications if we are dealing with buying/selling multiple stocks.
Comparison to other algorithms:
The dynamic programming algorithm is better than the brute-force approach, which would have an O(m^n) time complexity. An alternative to the dynamic programming algorithm is a greedy approach, which chooses the local optimal solution at each step and would be efficient with simpler cases. However, they're not guaranteed to make the globally optimal solution.

```
#include <iostream>
#include <vector>
#include <tuple>

using namespace std;
// Function to find optimal transactions to maximize profit
vector<tuple<int, int, int>> maximizeProfit(vector<vector<int>>& A, int k) {
    int m = A.size();
    int n = A[0].size();
    // Dynamic programming array to store maximum profit for each state
    vector<vector<vector<int>>> dp(m, vector<vector<int>>(n, vector<int>(k + 1, 0)));
    // Iterate over transactions
    for (int l = 1; l <= k; ++l) {
      // Iterate over days
       for (int j = 1; j < n; ++j) {
           int maxDiff = dp[0][j - 1][l - 1] - A[0][j - 1];
          // Iterate over stocks
           for (int i = 1; i < m; ++i) {
              // Update dp array based on the recurrence relation
              dp[i][j][l] = max(dp[i][j - 1][l], A[i][j] + maxDiff);
              // Update maxDiff for the current stock
              maxDiff = max(maxDiff, dp[i][j - 1][l - 1] - A[i][j - 1]);
```

```
            }
        }
    }

    int maxProfit = 0;
    tuple<int, int, int> result;

    for (int i = 0; i < m; ++i) {
        for (int l = 0; l <= k; ++l) {
            if (dp[i][n - 1][l] > maxProfit) {
                maxProfit = dp[i][n - 1][l];
                result = make_tuple(i + 1, 1, n);
            }
        }
    }

    return (maxProfit > 0) ? vector<tuple<int, int, int>>(1, result) : vector<tuple<int, int,
int>>();
}

int main() {
//Example matrix
    vector<vector<int>> A = {
        {1, 6, 8, 3},
        {2, 5, 7, 9},
        {12, 2, 6, 3}
    };

    int k = 2; // Set the maximum number of transactions

    // Find and print the optimal transactions
    vector<tuple<int, int, int>> result = maximizeProfit(A, k);

    if (result.empty()) {
        cout << "No profitable transactions." << endl;
    } else {
        cout << "Optimal transactions:" << endl;
        for (const auto& t : result) {
            cout << "(" << get<0>(t) << ", " << get<1>(t) << ", " << get<2>(t) << ")" << endl;
        }
```

```
    }

    return 0;
}
```

Task 7 C++ Implementation

Description of the implementation of the algorithm:

The code implements the pseudocode for the dynamic programming algorithm we developed in milestone 2. The pseudocode and the implementation below for the program has to take in the following:

-An input matrix A containing m stocks and n days to potentially buy and sell a stock

-A constant c that represents the cooldown period in days between buying and selling a stock.

The pseudocode and the implementation below for the program below has to output the following:

-A sequence of tuples (i,j,l) that yields the maximum potential profit by selling ith stock on lth day that was bought on jth day.

The implementation creates a data structure called “Interval” that holds the data of a buy sell interval that makes it easier to sort the intervals and later output them.

Limitations/Assumptions:

The list of intervals is assumed to already be sorted my sell day according to the pseudocode. Therefore, the execution of our implementation below in the main function starts at the for loop that calls the initializeP function.

Lessons learned/Comparative Analysis:

For us (the members of the group that worked on this part of the project), it was easier to reason about this dynamic programming problem from a top down perspective. This is because the solution to the problem can be phrased in this fashion:
“The optimal solution contains a list of intervals. If we sort the list of intervals by sell day (which makes sense because the quicker we can sell a stock,l the faster we can buy another stock), then the optimal list of intervals either includes the nth interval in the list of n intervals or it does not (because it would be more advantages (more profitable) to include a previous interval that overlaps with it.”
We believe this is because it is easier to think about the recursive solution then the iterative solution a bottom up approach would use.

Trade-Off Discussion

Due to this algorithm proceeding via m * n top-down recursive calls, its possible that the size of the stack used by the program could be exceeded. The advantage of the bottom up, iterative version of this dynamic programming problem is that the call stack does not grow with the input size due to execution proceeding via for loops instead of recursive calls.

```
#include <vector>
#include <iostream>
#include <unordered_map>
#include <algorithm>
using namespace std;

//Data structure representing a buy/sell interval
struct interval {
    int product;
    int buyDay;
    int sellDay;
    int profit;
    interval() {

    }

    interval(int newProduct, int newBuyDay, int newSellDay, int newProfit) {
            product = newProduct;
            buyDay = newBuyDay;
            sellDay = newSellDay;
            profit = newProfit;
    }
};
//This function intializes p which is a map from [interval indecies of the intervals]  -> [A pair holding the interval
void initializeP(unordered_map<int, pair<interval,int>>& p, int intervalIndex, vector<interval> &intervals, int c) {
    if(intervalIndex == 0)
    {
            return;
    }
```

```
    for(int i = intervalIndex - 1; i >= 0; i--) {
        interval intervalToCalculatePFor = intervals[intervalIndex];
        interval currentInterval = intervals[i];
        //If the sell day comes 2 days before the buy day
        if(currentInterval.sellDay  < intervalToCalculatePFor.buyDay - c)
        {
            pair<interval, int> newPair(currentInterval, i);
            p[intervalIndex] = newPair;
            return;
        }

    }
}

int getClosestIntervalIndex(int intervalIndex, unordered_map<int, pair<interval, int>>&
p) {
    if (p.find(intervalIndex) == p.end())
    {
        return -1;
    }



    return p.find(intervalIndex)->second.second;
}

int  opt(int intervalIndex, vector<interval> &intervals, unordered_map<int,int>&
memTable, unordered_map<int, pair<interval, int>>& p) {
    if(intervalIndex < 0)
    {
        return 0;
    }

    if(memTable.find(intervalIndex) != memTable.end() )
    {
        return memTable[intervalIndex];
    }
    else
    {
        //The optimal profit is equal to the maximum between:
```

```
        //The profit we can make at this current index in the optimal solution + the
optimal profit we can make at the
        //closest previous index that does not correspond to an interval that overlaps
with the interval at the current index or:

        //The profit we can make at just the previous index
        int maximum = max(intervals[intervalIndex].profit +
opt(getClosestIntervalIndex(intervalIndex, p), intervals, memTable, p), opt(intervalIndex
- 1, intervals, memTable, p));
        memTable[intervalIndex] = maximum;
        return maximum;
    }

}

void findSolution(int j, unordered_map<int, int>& memTable, vector<interval>& intervals,
unordered_map<int, pair<interval, int>>& p, vector<interval> &output) {
    if (j == -1)
    {
        return;
    }
    else
    {
        interval& currentInterval = intervals[j];
        if (intervals[j].profit + memTable[getClosestIntervalIndex(j, p)] >= memTable[j -
1])
        {
            output.push_back(currentInterval);
            findSolution(getClosestIntervalIndex(j, p), memTable, intervals, p, output);
        }
        else {
            findSolution(j-1, memTable, intervals, p, output);
        }
    }
}

bool comparisonFunction(interval& left, interval& right) {
    return left.sellDay < right.sellDay;
}
```

```
int main() {
    // c represents the cool down period in days between seling and buying another stock
    int c;
    cin >> c;
    //Map from interval indecies to a pair containing the previous interval index and the
interval at that previous index
    unordered_map<int, pair<interval, int>> p;

    vector<interval> intervals;
    unordered_map<int, int> memTable;
    vector<interval> output;

    //Example matrix of m products and n days to buy and sell products
     int arr[5][7] = { {2, 9, 8, 4, 5, 0, 7},
                       {6, 7, 3, 9, 1, 0, 8},
                       {1, 7, 9, 6, 4, 9, 11},
                       {7, 8, 3, 1, 8, 5, 2},
                       {1, 8, 4, 0, 9, 2, 1}};
     //This for loop assembles the vector of all valid intervals that are profitable in the
above matrix
    for (int i = 0; i < 5; i++) {
            interval temp;
            for (int j = 0; j < 7; j++) {
                    for(int k = j; k < 7; k++) {
                            temp.buyDay = j +1;
                             temp.sellDay = k +1;
                             temp.profit = arr[i][k] - arr[i][j];
                             temp.product = i + 1;
                             if(temp.profit > 0) {
                                     intervals.push_back(temp);
                             }
                    }
            }
    }

    //sort(intervals.begin(), intervals.end(), comparisonFunction);
    for (int i = 0; i < intervals.size(); i++) {
            initializeP(p, i, intervals, c);
    }
    int maxProfit = opt(intervals.size() - 1, intervals, memTable, p);
```

```
    findSolution(intervals.size() - 1, memTable, intervals, p, output);

    for (int i = output.size()-1; i >= 0; i--) {
        if(i!=0)
            cout << "(" << output[i].product << ", " << output[i].buyDay << ", " <<
output[i].sellDay << "), ";
        else
            cout << "(" << output[i].product << ", " << output[i].buyDay << ", " <<
output[i].sellDay << ")";
    }
}
```