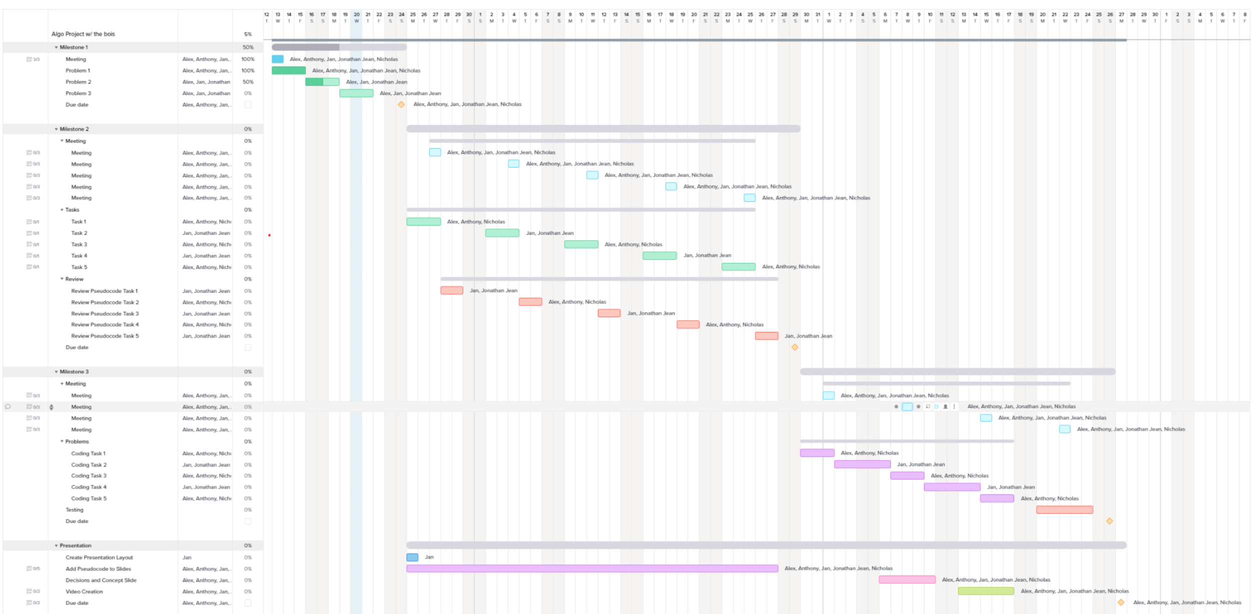
Project - Milestone 1

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

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

For stock with index 1:

- 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

/* 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);


}
```