COT5405 - Analysis of Algorithms
Programming Project

Team Members:

1.) Pavan Appikatla - 51971306
2.) Aditya Sure - 22413121
3.) Deepak Raju Ranga Raju - 70219269

In this project, we used java to design the required algorithms with the required time complexity and memory. Further running time comparisons are also included to help understand the time complexity.

Problem 1:

Alg 1:

Given a matrix A of m × n integers (non-negative) representing the predicted prices of m stocks for n days, find a single transaction (buy and sell) that gives maximum profit.

Alg1 Design a Θ(m ∗ n2) time brute force algorithm for solving Problem1

Input:

Prices for m stocks for n days -> matrix int[m][n]

Design of algorithm:

- Each stock record in the given matrix is considered.
- In each stock record maximum profit is calculated based on the following logic.
- Each day is considered a buy day and profit is calculated by considering each and every subsequent day as a sell day. For achieving this we used two loops one for traversing the buy day value and the nested loop for checking the maximum profit for all the other sell days.
- We are comparing each profit and storing the indexes of those days where maximum profit is received.
- The same logic is repeated for all the given number of stocks.

Correctness:

- The intuition in this algorithm is to find the minimum value day to be the buy day and the maximum possible day to be the selling day.
- Since, this is a brute force algorithm, we are considering each and every day as buy day and calculating the profit by considering each and every next day as sell day.
- While doing this process, the profit that is already calculated is compared with current profit. We will update our maximum profit achieved in this way.
- Since we are repeating the same process for M stocks and capturing the maximum profit along with the index of sell day and buy day we should end up with maximum profit stock and buy day and sell day indices.

Time complexity and Space Complexity:
We have used three nested loops to calculate profits in the program. Variable i = 0 to m, j = 0 to n and k = 0 to n. The time complexity works out to O(m*n*n).

Three variables are used to capture buy day, sell day and profit. Also iterative variables like i, j and k are used. Hence the time complexity is constant i.e., O(1).

```
maxProfit = 0;

Foreach stock data 'S' in M stocks
{
        For i = 0 to n-2
        {
                For j = i+1 to n-1
                {
                        If (maxProfit < S[j] – S[i])
                                maxProfit = S[j] – S[i]
                                buyIndex = i
                                sellIndex = j
                                stockIndex = Indexof(S)
                }
        }
}
Return stockIndex, buyIndex, sellIndex.
```

<u>Alg 2:</u>

Alg2 Design a $\Theta(m * n)$ time greedy algorithm for solving Problem1

Input:
Stock values for m stocks and n days.

Design of Algorithm:

- In this algorithm our intuition is to update the minimum value so far in a stock data simultaneously while we look for maximum profit by selecting next sell day.
- So we have avoided a nested for loop for selecting the sell day.
- While traversing through the day values of a stock, verify if the dayValue is less than the minimumSoFar we already stored.
- If it is less than the minimumSoFar, we will update minimumSoFar.
- While doing this, we will also compute profit by subtracting minimumSoFar from the current dayValue in the loop. We will update our profit with the maximum Value and also capture the day indices of buy day and sell day.
- We repeat the above steps for all the stocks thereby achieving maximum profit stock, buyDay and Sell day.

Correctness:

- The aim of this algorithm is to find the minimum value day to be the buy day and the maximum possible day to be the selling day.
- Since we have to compute maximum profit for a single stock in linear time, we just used a small logic to maintain minimumSoFar while simultaneously calculating profit for every possible day value.
- While doing this process, the profit that is already calculated is compared with current profit. We will update our maximum profit achieved in this way.
- Since we are repeating the same process for M stocks and capturing the maximum profit along with the index of sell day and buy day we should end up with maximum profit stock and buy day and sell day indices. Hence it gives us correct values for buy day, sell day and maximum profit stock index.

Time complexity and Space Complexity:
We are using two nested for loops i and j where i = 0 to m and j = 0 to n. Hence the time complexity will be O(m*n).
We used six variables to store maxindex, min index and profit. Hence the space complexity is constant i.e., O(1).

Algorithm:

```
For each stock data 'S' in M stocks
 {
        currentStockMinimum = 0
        for i = 0 to n-1
        {
                if(currentStockMinimum > S[i])
                {
                currentStockMinimum = S[i]
                buyDay = j
                }
                else if(currentStockMaximum < S[i] - currentStockMinimum)
                {
                currentStockMaximum = S[i] - currentStockMinimum
                sellDay = j
                stockIndex = i
                }
        }
        if(maxAcrossAllStocks < currentStockMaximum)
        maxAcrossAllStocks =  currentStockMaximum;
        finalBuyDayIndex = buyDay
        finalSellDayIndex = sellDay
        finalStockIndex=stockIndex

        }
Return stock, finalBuyDayIndex, finalSellDayIndex.
```

Alg3 Design a Θ(m ∗ n) time dynamic programming algorithm for solving Problem1

Using Memoization (TopDown):

Input:
Stock values for m stocks and n days.

Design of Algorithm:

- The algorithm is designed with an intuition that for every day we have two options i.e., whether to buy the stock or sell the stock.
- Buying a stock is possible only if we haven't buyed the stock earlier on any day.
- Selling a stock is possible only if we had already bought a stock earlier.
- Keeping these two conditions as possibilities for each day we are maintaining a variable buyFlag which gives the recursive sub problem an idea whether the stock is allowed to be bought or not.
- While doing this, we observe that we have many overlapping sub problems which we keep calculating again and again.
- For this, we have used a three dimensional array Cache to store the profit values that we have already calculated. (The three dimensional array holds m*n*2 profits. Here the last multiplication factor 2 is used because each day is considered as buy day as well as sell day while the algorithm is performed. Hence we get two values for each day)
- When the same subproblem is triggered in the program, we will first check the cache and return if the value is already calculated earlier.

Recursive Formulation:

// When the current day is considered as buy day

cache[mthShare][dayIndex][buyFlag=1] =

Maxof: (If we buy on this day or pass over the buying option to next day)

1. (-stockData[mthShare][dayIndex] + ComputeMaxShareValue(stockData, mthShare, 0, dayIndex + 1,cache) // Buying indicates a negative figure. Meaning we are expensing out to buy the stock.
2. ComputeMaxShareValue(stockData, mthShare, 1, dayIndex + 1,cache));

// When the current day is considered as sell day

cache[mthShare][dayIndex][buyableFlag=0] =

Maxof: (If we sell on this day or we may passover the sell option to the next day)

1. Math.max(+stockData[mthShare][dayIndex],

2. ComputeMaxShareValue(stockData, mthShare, 0, dayIndex + 1, cache));

By backtracking the cache[m][n][2] we will be able to capture the maximum profit and buyday and sell day indices.

Correctness:

- The aim in this algorithm is to find out the maximum profit from a given list of n day values.
- Here in the program, we traverse through each element of the stock data only once. However we use recursive calls to find out the optimal solution to the problem.
- At every step we choose maximum profit for both buy and sell possibilities of a day.
- The same steps are repeated for every stock and the values are stored in cache.
- By backtracking cache three dimensional array, we will achieve the maximum profit along with buy day and sell day indices.

Time Complexity and Space Complexity:
The time complexity of this algorithm is O(m*n). We are traversing through the stock data only once. However the recursive structure calls the same algorithm multiple times. But by the usage of memoization we have avoided the problem from becoming polynomial time on n.

We have used a three dimensional array of dimensions m*n*2. Hence, the space complexity of the program is O(m*n).

Challenges:
As we are computing the profit values by considering each day as buy day and sell day. We have to use a three dimensional array i.e, a buy version of 2D array and sell version of 2D array. Combining these two 2D arrays, we do backtracking and compute the required indices.

Algorithm:

```
ComputeMaxShareVal(S, Indexof(S), buyFlag, dayIndex, cache)
{
if (dayIndex == S.length)
{
                        return 0;
            }
            else
```

```
        {
                If (cache[Indexof(S)][dayIndex][buyableFlag] != 0)
{
                        return cache[Indexof(S)][dayIndex][buyableFlag]
}
else
{
                if (buyableFlag == 1)
{
                                return cache[Indexof(S)][dayIndex][buyableFlag] =
```

	-S[dayIndex] + ComputeMaxShareValue(S, Indexof(S), 0, dayIndex + 1,cache)
Maximum of	and
	ComputeMaxShareValue(S, Indexof(S), 1, dayIndex + 1,cache)

```
 else {

                return cache[Indexof(S)][dayIndex][buyableFlag] =
```

	shareMatrix[Indexof(S)][dayIndex]
Maximum of	and
	ComputeMaxShareValue(S, Indexof(S), 0, dayIndex + 1, cache));

```
        }
        }
}
```

```
Foreach stockdata S in M stocks
{
        FindMaximumof (ComputeMaxShareVal(S, Indexof(S), buyFlag, dayIndex, cache))

}
```

The values of buyDay and SellDay can be retrieved by back tracking the **cache** that is computed as part of the above algorithm.

To find out buyDay index:

Buy value is found out by traversing cache in reverse direction and wherever the maximum value is observed at buy flag = 1. That particular index represents buyDay.

To find out sellDay index:

Sell value is found by traversing cache in reverse direction and by deducting buy value obtained from above step from all values in cache where buy flag =0 and wherever it equals the maximumProfit.

Thus we return buyDay and sellDay indices.

Algo3B:

Bottom up implementation:

Input:
Stock values for m stocks and n days.

Design of Algorithm:

- The intuition in this algorithm is same as top down approach. However, instead of calculating subproblems when needed, we calculate the entire data starting from the first element and store it in a two dimensional array.

- The approach is to calculate profit[m][n] array completely which shows us the profit achieved as on nth day in that particular mth stock.
- We start from first day value in any stock and calculate the profit based on remember the past minimum value obtained and profit obtained and hence a dynamic approach of the algorithm
- In the profit[m][n] two dimensional array that we build, the last column gives the profit obtained in each stock.
- We will then backtrack the profit[][] to find out the maximum stock index, buyday and sell day indices.

Correctness:

- During each iteration in a particular stock data, we remember the solution for (i-1)th problem and use it for calculating ith problem.
- At every iteration, the minimum value till (i-1) and profit till (i-1) is stored in the profit array.
- Sequentially we use this minimum value and calculate maximum profit for the ith sub problem.
- Thereby we achieve maximum profit for each stock.

Time complexity and Space Complexity:
We have two nested for loops in the program i = 0 to m and j = 0 to n. Hence the time complexity of the program is O(m*n).

We have utilized profit[m][n] for storing the profit values. Hence the additional space complexity we used is O(m*n).

Algorithm:

```
Foreach stock data S in list of M stocks
{
                for i = 0 to m-1
        {
                    minvalue[Indexof(S)] = S[0]
            profit[Indexof(S)][0] = 0;

                        for int j=1 to n-1
                        {
```

```
                                    profit[Indexof(S)][j] =
Math.max(S[j]-minvalue[Indexof(S)],profit[Indexof(S)][j-1]);
                                    minvalue[Indexof(S)] = Math.min(minvalue[Indexof(S)],S[j]);
                            }
                    }

}
```

Now we have profit[][] matrix where the last row consists of maximum profits for each stock.

Pick the maximum value from profit[][n-1]. This gives us the maximum profit. Also, the maximum stock index can be obtained from this.

Now, we can pickup the minimum stock price and index that yielded the profit from minValue[] array.

The maximum stock price and its index can be achieved by traversing through the stock array where maximum profit is achieved in linear time.

Algo4:

Design a Θ(m ∗n^2k) time brute force algorithm for solving Problem2

Given:

Prices for m stocks for n days -> matrix int[m][n], k -> transactions

Design:

Initializing mToS[][] -> 0 this array contains the stock which has a maximum value between any buy and sell day.
Initializing max[][] -> 0 this array contains the maximum value between any two buy days and sell days.
Initializing profit to 0 -> contains the maximum profit yielded after iterating through m stocks.
We will be iterating across m stocks for each stock we will find all combinations of buy day and sell day and for each combination, we will recursively find all k transactions possible with those two pairs. If we have a sell day j and buy day j - 2 then we will recursively find the maximum profit achievable from j-3 days with k - 1 transaction. During this process, we will be maintaining maximums between any combination of buy and sell day and the stock associated with the maximum value for backtracking. We will update the global profit which will contain the

maximum profit so far and the associated transaction details like buy and sell days to print the transaction associated with the max profit.

```
findMaxProfitAlgo4(int[][] prices, int[][] max, int[][] mToS, int n, int k, int stock){
int profit = 0
for(int i = 1; i <= n; i++) {
 for (int j = i - 1; j >= 0; j--) {
Int temp = 0;
int t = prices[stock][i] - prices[stock][j];
if (t > max[j][i]) { max[j][i] = t; mToS[j][i] = stock; }
Int p1 = (TDR) findMaxProfitAlgo4(prices, max, mToS, j, k - 1, stock);
temp += p1 + max[j][i];
profit = max(profit, temp);
}}
return profit;
}
```

```
global_profit = 0;
for (int i = 0; i<stocks;i++)
max(global_profit, findMaxProfitAlgo4(int[][] prices, int[][] max, int[][] mToS, int n, int k, int stock))
```

We have used custom class TDR in our program to get the transaction sequence
TDR contains two attributes profit of type integer and comb of type string

```
TDR findMaxProfitAlgo4(int[][] prices, int[][] max, int[][] mToS, int n, int k, int stock) throws CloneNotSupportedException {
    if(k == 0)
       return new TDR();
    TDR profit = new TDR();
    for(int i = 1; i <= n; i++) {
       String comb = "";
       for (int j = i - 1; j >= 0; j--) {

          TDR temp = new TDR();
          int t = prices[stock][i] - prices[stock][j];
          if (t > max[j][i]) {
             max[j][i] = t;
```

```
            mToS[j][i] = stock;
          }
          if(max[j][i] > 0)
            comb = String.format("%s %s %s", mToS[j][i], j, i);
          TDR sub = (TDR) findMaxProfitAlgo4(prices, max, mToS, j, k - 1, stock).clone();
          temp.profit += sub.profit + max[j][i];
          temp.comb = comb + ((sub.comb != "")?( ((comb == "")?"":",") + sub.comb):"");
          profit = (TDR)Max.compare(temp, profit).clone();
        }
      }
      return profit;

TDR profit = new TDR();
int[][] mToS = new int[prices[0].length][prices[0].length];
      int[][] max = new int[prices[0].length][prices[0].length];
      for(int i = 0; i < prices.length; i++){
        profit = Max.compare(profit, findMaxProfitAlgo4(prices, max, mToS, prices[0].length-1, k,
i));
      }
Combination -> profit.comb
Total-profit -> profit.profit
```

Correctness:

Since we are finding the best k transaction for every possible buy and sell day and maintaining the maximum between any sell day and buy day we will be able to get the maximum profit possible across m stocks for k transactions. Therefore this solution will lead to the maximum profit with k transactions within the given time complexity.

Time complexity:

O(m * n^(2K)) -> for each n^2 buy and sell day at kth transaction there will be k-1 transaction with the remaining days.

Space complexity:

O(n ^ 2) to backtrack the indices and maintain the maximum.

Design a Θ(m ∗n^2∗k) time dynamic programming algorithm for solving Problem2

Algo 5:

Given:

Prices for m stocks for n days -> prices int[m][n],
k -> max transactions possible

Design:

Int[][] dp -> maintains maximum profit achieved any day for k transaction
String[][[] store -> maintains the pairs of transactions for the maximum profit achieved any day for k transaction.

We will be iterating through k transactions and for each transaction k we will iterate across all the stocks for each stock we will be iterating through all the n days for each day we will iterate through all possible buys days, for each pair we will find the profit associated with that pair and use our DP table to get the max profit possible from buy day with k - 1 transaction. dp[i][j] represents max possible profit achieved with i transactions from day j. We make use of the below recurrence relation to solve the problem.

```
for (int i = 1; i <= k; i++) {
        for(int v = 0; v < stocks; v++) {
           for (int j = 1; j < n; j++) {
              int maxFactor = 0;
              for (int m = 0; m < j; m++) {
                 int t = arr[v][j] - arr[v][m] + dp[i - 1][m];
                 maxFactor = Math.max(maxFactor, t);
              }
              int temp = Math.max(dp[i][j - 1], maxFactor);
              dp[i][j] = Math.max(temp, dp[i][j]);
           }
        }
     }
```

We will use String[][] to store the sequence in the same order as we update the profit.

```
// array to store the maximum profit achievable with k transactions across m stocks for n days
     int[][] dp = new int[k + 1][n + 1];
     // array to store the transaction combination
     String[][] store = new String[k+1][n+1]

     for (int i = 1; i <= k; i++) {

        for(int v = 0; v < stocks; v++) {
           for (int j = 1; j < n; j++) {
```

```
            int maxFactor = 0;
            String comb = "";

            for (int m = 0; m < j; m++) {
               int t = arr[v][j] - arr[v][m] + dp[i - 1][m];
               if(t > maxFactor)
                  comb = String.format("%s %s %s", v, m, j) + ((store[i-1][m] !=
""")?","+store[i-1][m]:"" );
               maxFactor = Math.max(maxFactor, t);
            }

            if(dp[i][j-1] > maxFactor)
               comb = store[i][j-1];
            int temp = Math.max(dp[i][j - 1], maxFactor);
            if(temp < dp[i][j])
               comb = store[i][j];
            dp[i][j] = Math.max(temp, dp[i][j]);
            store[i][j] = comb;
         }

      }
   }
```

Recurrence relation:

dp[k, j] - { max(max (dp[k][j-1], max 0 <= m < j (prices[stock][j] - prices[stock][m]) + dp[k-1][m]), opt[k][j]), k >= 0 and j >= 0

{ 0, k <= 0

{ 0, j <= 0

k - transaction
j - day
Stock -> stock index

Correctness:

This problem has an optimal substructure so it can be solved using dynamic programming. Since we are maintaining the maximum profit for each transaction starting from 1 to k across all m stocks for all days we will be able to get the maximum profit achievable on the n the day with

k transaction. Base cases are where i=0, which indicates there were 0 transactions and the maximum profit associated with this is 0. We will check for each selling day the best buy day where we can maximize the profit, if the total profit is greater than the corresponding jth day with k transaction then we will update the DP table, if we don't buy then we will consider the dp[k][j-1] profit and update it accordingly. In this way, we will make sure that correctness is achieved.

Complexity:
Time complexity: O(m* n ^ 2 * k)
Space complexity: O(k * n)

Design a $\Theta(m * n * k)$ time dynamic programming algorithm for solving Problem2
Given
(6A) Memoization:
Design:
We utilized recursive calls and memoization to compute the highest profit with a maximum of k transactions.

There are 3 possible paths in the recursion either you can (buy or sell) or you can skip. For each subproblem, you will have only two possibilities either to (buy or sell) depending upon the parent problem or you can just skip.
If the profit for the subproblem already exists in the memoization matrix return it.
else
Find the profit obtained by skipping the day irrespective of state.
if(brought) then there are three possibilities {
Sell the stock on that i day.
(stock here corresponds to the current stock index)
1.) Find a profit that is maximum by buying the stock on the ith day for stock+1
2.) Find a maximum profit by buying at ith day for stock -1.

3.) Find the maximum profit by buying the stock on the ith day.
4.) Find the max of all the profit with the skip profit
}
else{
1.) Find the profit of buying the stock on the ith day for the current stock.
2.) Find the profit of buying the stock on the ith day for the stock+1.
3.) Find the profit of buying the stock on the ith day for the stock-1.
4.) Get the max of all the stocks along the skip profit and return it.
}

Store the result for the sub-problem in a 2D memoization matrix or hashmap, we are using a custom TDR class for backtracking.

```
public TDR maximizeProfitAlgo6TopDown(int[][] arr, int i, int k, boolean buy, HashMap<String,
TDR> memo, int stock, int buyIndex, int prevStock) {
      if (i >= arr[0].length || k == 0 || stock >= arr.length || stock < 0) return new TDR();
      String key = String.format("%s-%s-%s-%s", i, k, buy, stock);
      if(!memo.containsKey(key)) {

         subProblems++;
         TDR profit = this.maximizeProfitAlgo6TopDown(arr, i + 1, k, buy, memo, stock,
buyIndex,-1);
         if (buy) {
            TDR ps = this.maximizeProfitAlgo6TopDown(arr, i, k - 1, false, memo, stock, -1,-1);
            TDR pu = this.maximizeProfitAlgo6TopDown(arr, i, k - 1, false, memo, stock + 1, -1,-1);
            TDR pd = this.maximizeProfitAlgo6TopDown(arr, i, k - 1, false, memo, stock - 1, -1,-1);
            TDR temp  = Max.compare(Max.compare(ps, pu), pd);
            temp.profit = temp.profit + arr[stock][i];
            temp.sellDay = i ;
            profit = (TDR)Max.compare(profit, temp).clone();

         } else {
            TDR ps = this.maximizeProfitAlgo6TopDown(arr, i + 1, k, true, memo, stock, i,-1);
            TDR pu = new TDR();
            TDR pd = new TDR();

            if(prevStock != stock + 1)
               pu = this.maximizeProfitAlgo6TopDown(arr, i, k, false, memo, stock + 1, -1,-1);
            if(prevStock != stock - 1)
               pd = this.maximizeProfitAlgo6TopDown(arr, i, k, false, memo, stock-1, -1, stock);

            ps.profit = ps.profit - arr[stock][i];
            ps.comb = String.format("%s %s %s", stock, i, ps.sellDay) + ((ps.comb !=
"")?(","+ps.comb):"");
            profit = Max.compare(profit, ps);
            profit = Max.compare(profit, pu);
            profit = Max.compare(profit, pd);
         }
         memo.put(key, profit);
      }
```

Recursive relation:

1.) MP(A, i, k, b, s, n, m) - max(A[s][i] + max(MP(A, i, k-1, 0, s, n, m), MP(A, i, k-1, 0, s+1, n, m), MP(A, i, k-1, 0, s-1, n, m)), MP(A, i+1, k-1, b, s, n, m)), b == 1
2.) max(-A[s][i] + max(MP(A, i, k-1, 1, s, n, m), MP(A, i, k-1, 0, s+1, n, m), MP(A, i, k-1, 0, s-1, n, m)), MP(A, i+1, k-1, b, s, n, m)) b == 0
3.) 0 - k== 0
4.) 0 - i > n
5.) 0 - stock < 0
6.) 0 - stock > m

k - denotes the transaction
A - 2d array containing prices for m stocks
i - index of the day
b - buy or sell flag
s - stock index
n - number of days
m - number of stocks

Correctness:
The problem has an optimal substructure, so it can be solved using dynamic programming.
In the above recursive call, we are finding all possible combinations to find the k transactions to maximize the profit. With the help of memoization, we are curtailing the exponential time complexity making it polynomial.

Time Complexity:
Total time complexity is O(m * n * k) when we used a global variable to keep track of the number of subproblems we came to know that there are 3 * m * n * k subproblems ignoring the constant 3 we are getting the recommended time complexity.

Space complexity:
Since we store the profit values of m * n * k subproblems
O(m * n * k)

Algo 6:
(6B) Bottom-up dynamic programming
Design a $\Theta(m * n * k)$ time dynamic programming algorithm for solving Problem2
This is the same as the algorithm (5) but won't check for all possible buy days for a sell day.
Given:
Prices for m stocks for n days -> prices int[m][n], k -> transactions possible scalar
Definition:
Int[][] dp -> maintains maximum profit achieved any day for k transaction
String[][[] store -> maintains the pairs of transactions for the maximum profit achieved any day for k transaction.

We will be iterating through k transactions and for each transaction k we will iterate across all the stocks for each stock we will be iterating through all the n days for each day we won't iterate through all possible buys days, we will maintain a max-factor which will have the maximum of the comparison factor. For example, if there are n days, To find the maximum profit at the jth day with k transactions according to the earlier recurrence relation for the problem 5 profit[k][j] = price[j] - price[m] + profit[k-1][m] (m is between 0 to j). Here instead of iterating through all j-1 days for each j, we will maintain a variable that will maintain the maximum of this factor (-price[m] + profit[k-1][m]) so we need not iterate back.

```
ffor (int i = 1; i <= k; i++) {
        for (int v = 0; v < stocks; v++) {
           int maxFactor = Integer.MIN_VALUE;

           int maxIndex = -1;
           for (int j = 1; j < n; j++) {
              int t = dp[i - 1][j - 1] - arr[v][j - 1];
              if(t > maxFactor) {
                 maxIndex = j-1;
              }

               maxFactor = Math.max(maxFactor, t);
               int temp = Math.max(dp[i][j - 1], arr[v][j] + maxFactor);
              dp[i][j] = Math.max(dp[i][j], temp);
           }
        }
```

We are maintaining String[][] to store the sequence in the same way when we update the profit

```
 String store[][] = new String[k+1][n+1];
      int dp[][] = new int[k + 1][n + 1];

      for (int i = 1; i <= k; i++) {
         for (int v = 0; v < stocks; v++) {
            int maxFactor = Integer.MIN_VALUE;

            String comb = "";
            int maxIndex = -1;
```

```
            String maxComb = "";
            for (int j = 1; j < n; j++) {
               int t = dp[i - 1][j - 1] - arr[v][j - 1];
               if(t > maxFactor) {
                  maxIndex = j-1;
                  maxComb = store[i-1][j-1];
               }

               comb = String.format("%s %s %s", v, maxIndex, j) +
((maxComb!="")?(","+maxComb):"") ;

               maxFactor = max(maxFactor, t);
               if(arr[v][j] + maxFactor < dp[i][j-1]){
                  comb = store[i][j-1];
               }
               int temp = max(dp[i][j - 1], arr[v][j] + maxFactor);
               if(dp[i][j] > temp){
                  comb = store[i][j];
               }
               dp[i][j] = max(dp[i][j], temp);
               store[i][j] = comb;
            }
         }
      }
```

Recurrence relation:

dp[k][j] = 0 k == 0
0 j == 0
max(dp[k][j], dp[k][j-1], A[v][j] + max(dp[k-1][j-1] - A[v][j-1], max-factor));
otherwise

k - transaction
v - stock index
j - sell day
max-factor -> will maintain max(dp[k-1][j-1] - A[v][j-1]) from j to 0

Correctness:

This problem has an optimal substructure so it can be solved using dynamic programming.

Since we are maintaining the maximum profit for each transaction starting from 1 to k across all m stocks for all days we will be able to get the maximum profit achievable on the n the day with k transaction. Base cases are where i=0, which indicates there were 0 transactions and the maximum profit associated with this is 0. We will check for each selling day the best buy day at which we can maximize the profit, if the total profit is greater than the corresponding jth day with k transaction then we will update the DP table, if we don't buy then we will consider the dp[k][j-1] profit and update it accordingly. In this way, we will make sure that correctness is achieved.

Time complexity:

Maintaining the max-factor reduced one inner loop which reduced the time complexity from algo 5 to O(m * n * k).

Space complexity:

o(k * n) -> to store the maximum value of stock and the buy and sell day responsible for the profit.

Algo 7:

Design:

Using the brute force strategy, we try every combination to gain the most profit out of "m" stocks and "n" days in order to solve this problem. We have two options each day: either we buy or sell on that day. If we acquire a stock on day i then we can sell it from "i+1" to "n" . If we sell a stock on day i we can only acquire the following stock after day "i+c+1."
To find the answer, we are using m*2^n recursive calls. Every day offers us two options.

Function algo-7 will return maximum profit from day i to n for a given stock.
If we bought some stock in some day then we will find the maximum between two calls, either we will skip the day or sell it and buy the next stock at i + c+ 1th day.
If we didn't bought the stock then there will be two possibilities, either skip the day or buy the stock in the next day.
max(algo7(false, i + 1, n, arr, c, max-arr, stock), algo7(true, i + 1, n, arr, c, max-arr, stock))

```
algo7(boolean brought, int i, int n, int[] arr, int c, int[] max, int stock) {
    if(i >= n)
       return 0;
    int profit = algo7(brought, i+1, n, arr, c, max, stock);
    if(brought) {
       int temp =algo7(false, i + c + 1, n, arr, c, max, stock);
```

```
            temp += arr[i];
            profit = max(profit, temp);
            return profit;
        }
        else {
            int ps = algo7(true, i + 1, n, arr, c, max, stock);
            profit = ps - arr[i];
            profit = max(ps, profit);
        }
        max[i] = max(profit, max[i]);
        return max[i];
    }

global_profit = 0
 for (int i = 0; i < stockPrices.length; i++)
              global_profit = Max.compare(global_profit, algo7(false, 0, stockPrices[0].length,
stockPrices[i], c, opt, i));
}
```

We are using class TDR instead of integer in the program to maintain the transaction sequence in the program

```
TDR algo7(boolean brought, int i, int n, int[] arr, int c, TDR[] max, int stock) throws
CloneNotSupportedException {
        if(i >= n)
            return new TDR();

        subProblems++;
        TDR profit = algo7(brought, i+1, n, arr, c, max, stock);
        if(brought) {
            TDR temp = algo7(false, i + c + 1, n, arr, c, max, stock);
            temp.profit += arr[i];
            temp.sellDay = i;
            profit = Max.compare(profit, temp);
            return profit;
        }
        else {
            TDR ps = algo7(true, i + 1, n, arr, c, max, stock)
```

```
            ps.profit = ps.profit - arr[i];
            ps.comb = String.format("%s %s %s", stock, i, ps.sellDay) + ((ps.comb !=
"")?(","+ps.comb):"");
            profit = Max.compare(ps, profit);
        }
        max[i] = Max.compare(profit, max[i]);
        return max[i];
    }
```

.

Recurrence relation:

algo7(brought, i, n, arr, c, max, stock) = 0 if i>=n

max(algo7(true, i + 1, n, arr, c, max-arr, stock), algo7(false, i + c + 1, n, arr, c, max-arr, stock)) if brought == 1

max(algo7(false, i + 1, n, arr, c, max-arr, stock), algo7(true, i + 1, n, arr, c, max-arr, stock)) otherwise

Correctness:

If i >= n, then the base case should result in 0. We can simply obtain the overall maximum profit across the full array since we are identifying all feasible combinations of buy, sell day, and skip for all the stocks from 0 to m. Recursive calls of "m*2n" can be made using recursion. In order to determine whether the current profit is greater than the obtained, we can obtain the maximum profit after each recursive call. This demonstrates that we can find the best solution.

Complexity:

Time Complexity - O(m * (2^n)) 2^n possibilities across m stocks

Space Complexity - O(n) -> maximum value at every day across m stocks

ALG 8 Design a Θ(m ∗ n^2) time dynamic programming algorithm for solving Problem3

Design:

Given:

stockPrices -> m * n matrix

Scalar value c

Int[][] max -> array which will maintain the maximum between any two indexes

Definition:

We will maintain a max array which will store the maximum profit between any buy and sell days. We will iterate across every stock and for each stock, for every day we will find the all possible buy days from the sell day, during this iteration we will compare the profit between sell day and all possible buy days with max[sell day][buy day], if it is greater we will update.
So if we want to find the maximum profit achievable at the jth day, then we will loop through the buy days from j-1 to 0, the profit we will calculated as max[sell day][buy day] + max[buy day - (c+1))] and it will get updated into max[sell day][0] if it is greater than max[sell day][0].

```
algo8(int[][] stockPrices, int c,  int[][]max, String[][] store) {
      for(int s = 0; s < stockPrices.length; s++){
         max[0][0] = 0;
         int t = stockPrices[s][1] - stockPrices[s][0];
         max[1][0] = Math.max(max[1][0], t);

         for(int i = 2; i < stockPrices[0].length; i++){
              max[i][0] = Math.max(max[i][0], max[i-1][0]);

            for(int j = i-1; j>=0; j--) {
               t = stockPrices[s][i] - stockPrices[s][j];
                max[i][j] = Math.max(t, max[i][j]);
               int prevIndex = j - (c + 1);
               max[i][0] = Math.max(max[i][j] + (( prevIndex < 0) ? 0 : max[prevIndex][0]),
max[i][0]);
            }
         }
      }
      return max[stockPrices[0].length-1][0];
   }
```

We are using String[][] to store the sequences in the program.

```
for(int s = 0; s < stockPrices.length; s++){
         max[0][0] = 0;
         int t = stockPrices[s][1] - stockPrices[s][0];
         if(t > max[1][0])
            store[1][0] = String.format("%s %s %s", s, 0, 1);
         max[1][0] = Math.max(max[1][0], t);

         for(int i = 2; i < stockPrices[0].length; i++){
```

```
            int temp = 0;
            String comb = "";
            if(max[i-1][0] > max[i][0])
               store[i][0] = store[i-1][0];
            max[i][0] = Math.max(max[i][0], max[i-1][0]);

            for(int j = i-1; j>=0; j--) {
               t = stockPrices[s][i] - stockPrices[s][j];
               if (t > max[i][j]) {
                  comb = String.format("%s %s %s", s, j, i);
                  store[i][j] = comb;
               }

               max[i][j] = Math.max(t, max[i][j]);
               int prevIndex = j - (c + 1);
               if(max[i][j] + (( prevIndex < 0) ? 0 : max[prevIndex][0]) > max[i][0] ) {
                  store[i][0] = store[i][j] + ((prevIndex < 0) ? "" : ((store[prevIndex][0] != "") ? ("," +
store[prevIndex][0]) : ""));
               }
               max[i][0] = Math.max(max[i][j] + (( prevIndex < 0) ? 0 : max[prevIndex][0]),
max[i][0]);
            }

         }

      }
```

Recurrence relation:

max[j][0] = 0; j == 0

0<=m<j max(max(j, m) + max(m - (c+1),0)); otherwise

j - day-index

c - restricted days

Correctness:

After iterating across m-1 th stock, we are confident that we will be maintaining the max profit across all possible buy and sell day till m-1th stock. At mth stock for the sell day lets say n we will find the best ith buy day, if we find the profit obtained is greater than the max maintained

in the array, we will update. Since we are maintaining maximum profit across all possible indexes over m stocks, we should be able to get the maximum profit at the day n across m stocks with the restricted parameter c.

Complexity:

Time complexity: O(m * (n ^2)) for all m stocks we are maintaining the max of all possible indexes so n^2.

Spacecomplexity: O(n ^ 2) since we are maintaining the max value across all possible indexes

ALG 9 Design a $\Theta(m * n)$ time dynamic programming algorithm for solving Problem3

(9A) Memoization:

We utilized recursive calls and memoization to compute the highest profit with a cool down of c days.

There are 3 possible paths in the recursion either you can (buy or sell) or you can skip. For each subproblem, you will have only two possibilities either to (buy or sell) depending upon the parent problem or you can just skip.
If the profit for the subproblem already exists in the memoization matrix return it.
else
Find the profit obtained by skipping the day irrespective of state.
if(brought) then there are three possibilities {

Sell the stock on that i day.
(stock here corresponds to the current stock index)

5.) Find a profit that is maximum by buying the stock on the i + (c+1) th day for stock+1
6.) Find a maximum profit by buying at (i + c+ 1) th day for stock -1.
7.) Find the maximum profit by buying the stock on the i + (c+1) th day.
8.) Find the max of all the profit with the skip profit

}
else{

5.) Find the profit of buying the stock on the ith day for the current stock.
6.) Find the profit of buying the stock on the ith day for the stock+1.
7.) Find the profit of buying the stock on the ith day for the stock-1.
8.) Get the max of all the stocks along the skip profit and return it.

}
Store the result for the sub-problem in a 2D memoization matrix or hashmap
We are using custom class to get the sequences

Recursive relation:

7.) MP(A, i, c, b, s, n, m) - max(A[s][i] + max(MP(A, i + c +1, c, 0, s, n, m), MP(A, i + c+ 1, c, 0, s+1, n, m), MP(A, i +c+1, c, 0, s-1, n, m)), MP(A, i+1, c, b, s, n, m)), b == 1

8.) max(-A[s][i] + max(MP(A, i, c, 1, s, n, m), MP(A, i, c, 0, s+1, n, m), MP(A, i, c, 0, s-1, n, m)), MP(A, i+1, c, b, s, n, m)) b == 0

9.) 0 - i > n

10.) 0 - stock < 0

11.) 0 - stock > m

c - denotes the cool down period
A - 2d array containing prices for m stocks
i - index of the day
b - buy or sell flag
s - stock index
n - number of days
m - number of stocks

We are using custom class TDR to get the maximum profit and the transaction details

```
TDR algo9TopDown(int[][] arr, int i, int k, boolean buy, HashMap<String, TDR> memo, int stock, int buyIndex, int prevStock) throws CloneNotSupportedException {

    //base condition

    if (i >= arr[0].length ||  stock >= arr.length || stock < 0) return new TDR();
    String key = String.format("%s-%s-%s", i, buy, stock);
    // if the key is present in the DP table don't process

    if(!memo.containsKey(key)) {
       subProblems++;
       //skip the day

       TDR profit = this.algo9TopDown(arr, i + 1, k, buy, memo, stock, buyIndex,-1);
       if (buy) {
          TDR ps = this.algo9TopDown(arr, i + k + 1, k ,false, memo, stock, -1,-1);
          TDR pu = this.algo9TopDown(arr, i + k + 1, k, false, memo, stock + 1, -1,-1);
          TDR pd = this.algo9TopDown(arr, i + k + 1, k, false, memo, stock - 1, -1,-1);
          TDR temp  = Max.compare(Max.compare(ps, pu), pd);
          temp.profit = temp.profit + arr[stock][i];
          temp.sellDay = i ;
          profit = (TDR)Max.compare(profit, temp).clone();

       } else {
          TDR ps = this.algo9TopDown(arr, i + 1, k, true, memo, stock, i,-1);
          TDR pu = new TDR();
          TDR pd = new TDR();
          if(prevStock != stock + 1)
             pu = this.algo9TopDown(arr, i, k, false, memo, stock + 1, -1,-1);
          if(prevStock != stock - 1)
```

```
                pd = this.algo9TopDown(arr, i, k, false, memo, stock-1, -1, stock);

            ps.profit = ps.profit - arr[stock][i];
            ps.comb = String.format("%s %s %s", stock, i, ps.sellDay) + ((ps.comb !=
"")?(","+ps.comb):"");
            //compare among other profits
            profit = Max.compare(profit, ps);
            profit = Max.compare(profit, pu);
            profit = Max.compare(profit, pd);
          }
          memo.put(key, profit);
      }

      return (TDR)memo.get(key).clone();
   }
```

Correctness:
The problem has an optimal substructure, so it can be solved using dynamic programming.
In the above recursive call, we are finding all possible combinations to find the maximum profit obtained with c cool down period to maximize the profit. If there is no memoization it will take O(m * 4 ^ n) With the help of memoization, we are curtailing the exponential time complexity making it polynomial. We can solve the'm*n' subproblems using recursion. In order to determine whether the current profit is more than the gained, we can obtain the maximum profit after each recursive call. This shows that we are utilizing a recursive call to check all potential subproblems and then using memoization to get the best solution.

Complexity:
Time-complexity -
We had 3 * m * n total subproblems to solve so the time complexity is O(m * n)
Space-Complexity:
The memoization dictionary stores the profit achieved for every subproblem which is O(m*n)

9B.)

Definition:
For the bottom up approach, we will be storing two subproblems for every day across m stocks. For v at the n-1 day, we will store the profit associated with v and n-1 if we didn't buy the stock on the n-1 day and profit associated with v and n-1 if we brought the stock. These two values will be stored in the hashmap. By utilising this information we can find the profit associated with the index 0 after iterating through the m stocks where we didn't buy any stock.
The dp[v, n-1, true] and dp[v, n-1, false] can be found using the below recurrence relation

```
Int dp[][][]
```

```
arr -> prices array
 int m = arr.length-1;
 int n = arr[0].length-1;
 for(int i = n; i>=0; i--){
    for(int v = 0; v<=m; v++){
       int skip = dp[v][i+1][1];
       int pu = dp[v+1][i+c+1][0];
       int ps = dp[ v][i+c+1][ 0]
       int pd = dp[v-1][i+c+1][ 0];
       dp[v][i][1] = max(max(max(ps, pd), pu) + arr[v][i], skip));

       skip = dp[ v][i+1][ 0]
       ps = dp[v][i][1] - arr[v][i];
       pd = dp[ v-1][i] [0]
       dp[v][i][0] = max(skip, max(ps, pd)));

    }
 }
```

We are using custom class TDR in the program to backtrack and find the sequences

```
HashMap<String, TDR> dp = new HashMap<String, TDR>();
for(int i = n; i>=0; i--){
        //iterate through m days
        for(int v = 0; v<=m; v++){
           String bK = String.format("%s %s %s", v, i, true);
           String sK = String.format("%s %s %s", v, i, false);
           TDR skip = dp.getOrDefault(String.format("%s %s %s", v, i+1, true), new TDR());
           TDR pu = dp.getOrDefault(String.format("%s %s %s", v+1, i+c+1, false), new TDR());
           TDR ps = dp.getOrDefault(String.format("%s %s %s", v, i+c+1, false), new TDR());;
           TDRpd = dp.getOrDefault(String.format("%s %s %s", v-1, i+c+1, false), new TDR());
           TDR t = Max.compare(Max.compare(ps, pd), pu);
           t.sellDay = i;
           t.profit += arr[v][i];
           dp.put(bK, Max.compare(t, skip));
           skip = dp.getOrDefault(String.format("%s %s %s", v, i+1, false), new TDR());
           ps = (TDR) dp.getOrDefault(String.format("%s %s %s", v, i, true), new TDR());
ps.profit += -arr[v][i];
           if(ps.sellDay != i)
              ps.comb = String.format("%s %s %s", v, i, ps.sellDay) + ((ps.comb !=
"")?(","+ps.comb):"");
           pd = dp.getOrDefault(String.format("%s %s %s", v-1, i, false), new TDR());
           dp.put(sK, Max.compare(skip, Max.compare(ps, pd)));

        }}
```

Recurrence relation:

dp[v, i,buy] = 0 if i < 0, v < 0

max(max(dp[v+1, i+c+1, 0], dp[v, i+c+1, 0], dp[v-1, i+c+1, 0]) + prices[v][i], dp[v, i+1, 1)) if buy == 1

max(max(dp[v, i, 1], dp[v-1, i, 0]), dp[v, i+1, 0]) if buy == 0;

v - stockindex

i - day index

buy - flag which has two values false or true

Correctness:

We are maintaining maximum profit obtained for the two possibilities sold and not sold for every day in n across m stocks. Initially, we are starting from the n day as we will move towards lower values of n then finally, we will be able to find the max profit obtained starting at the 0th day with the buy flag false. In this way, we are checking all possibilities to maximize the profit with the cool period c.

Complexity:

Time complexity will be O(m * n).

Space complexity: O(m * n).

Contributions:

1.) Pavan Aapikatla: Worked on the design and analysis of algorithms, coding, testing, and report.

2.) Adithya: Worked on the design and analysis of algorithms, coding, and experimental study.

3.) Deepakraju Rangaraju: Worked on the design and analysis of algorithms, coding, testing and report.

Observations/Conclusions:

1.) For the bonus question, TASK-9A and TASK-9B ran faster than TASK-8 when there is no backtracking, after adding backtracking TASK-9A and TASK-9B took more time to execute than TASK-8. Moreover, we think that the clone operation is very costly, maybe that might have slowed down the DP. Without backtracking Task-9A, Task-9B is running more quickly than the other approaches.

2.) We faced challenges in getting the transaction details (buy and sell day for algo 6 top down) so we have used a class TDR which will contain the transaction information. For the bonus question, o ((n ^ 2) * m) was running faster than the o(n * m) solution because of backtracking, the backtracking will try to clone the value present in the hashmap to prevent overriding different values present in the subproblems. (value present is a class object) This clone operation is taking more time than expected.

3.) For the program 6A it gave us 3 * m * n * k subproblems, though it is asymptotically the same, it had so many subproblems. For programs 9A and 3A, it took 3 * m * n subproblems.